The Science of Computing

The Art of High Performance Computing, volume 1

Evolving Copy - open for comments

Victor Eijkhout
with
Edmond Chow, Robert van de Geijn

3rd edition 2022, formatted January 15, 2026
Series reference: https://theartofhpc.com
This book is published under the CC-BY 4.0 license.



Introduction to High-Performance Scientific Computing © Victor Eijkhout, distributed under a Creative Commons Attribution 4.0 Unported (CC BY 4.0) license and made possible by funding from The Saylor Foundation http://www.saylor.org.

Preface

The field of high performance scientific computing lies at the crossroads of a number of disciplines and skill sets, and correspondingly, for someone to be successful at using high performance computing in science requires at least elementary knowledge of and skills in all these areas. Computations stem from an application context, so some acquaintance with physics and engineering sciences is desirable. Then, problems in these application areas are typically translated into linear algebraic, and sometimes combinatorial, problems, so a computational scientist needs knowledge of several aspects of numerical analysis, linear algebra, and discrete mathematics. An efficient implementation of the practical formulations of the application problems requires some understanding of computer architecture, both on the CPU level and on the level of parallel computing. Finally, in addition to mastering all these sciences, a computational scientist needs some specific skills of software management.

While good texts exist on numerical modeling, numerical linear algebra, computer architecture, parallel computing, performance optimization, no book brings together these strands in a unified manner. The need for a book such as the present became apparent to the author working at a computing center: users are domain experts who not necessarily have mastery of all the background that would make them efficient computational scientists. This book, then, teaches those topics that seem indispensible for scientists engaging in large-scale computations.

This book consists largely of theoretical material on HPC. For programming and tutorials see the other volumes in this series. The chapters in this book have exercises that can be assigned in a classroom, however, their placement in the text is such that a reader not inclined to do exercises can simply take them as statement of fact.

Public draft This book is open for comments. What is missing or incomplete or unclear? Is material presented in the wrong sequence? Kindly mail me with any comments you may have.

You may have found this book in any of a number of places; the authoritative location for all my textbooks is `https://theartofhpc.com`. That page also links to `lulu.com` where you can get a nicely printed copy.

Victor Eijkhout `eijkhout@tacc.utexas.edu`
Research Scientist
Texas Advanced Computing Center
The University of Texas at Austin

Acknowledgement Helpful discussions with Kazushige Goto and John McCalpin are gratefully acknowledged. Thanks to Dan Stanzione for his notes on cloud computing, Ernie Chan for his notes on scheduling of block algorithms, and John McCalpin for his analysis of the top500. Thanks to Elie de Brauwer, Susan Lindsey, Tim Haines, and Lorenzo Pesce for proofreading and many comments. Edmond Chow wrote the chapter on Molecular Dynamics. Robert van de Geijn contributed several sections on dense linear algebra.

Introduction

Scientific computing is the cross-disciplinary field at the intersection of modeling scientific processes, and the use of computers to produce quantitative results from these models. It is what takes a domain science and turns it into a computational activity. As a definition, we may posit

> The efficient computation of constructive methods in applied mathematics.

This clearly indicates the three branches of science that scientific computing touches on:

- Applied mathematics: the mathematical modeling of real-world phenomena. Such modeling often leads to implicit descriptions, for instance in the form of partial differential equations. In order to obtain actual tangible results we need a constructive approach.
- Numerical analysis provides algorithmic thinking about scientific models. It offers a constructive approach to solving the implicit models, with an analysis of cost and stability.
- Computing takes numerical algorithms and analyzes the efficacy of implementing them on actually existing, rather than hypothetical, computing engines.

One might say that 'computing' became a scientific field in its own right, when the mathematics of real-world phenomena was asked to be constructive, that is, to go from proving the existence of solutions to actually obtaining them. At this point, algorithms become an object of study themselves, rather than a mere tool.

The study of algorithms became especially important when computers were invented. Since mathematical operations now were endowed with a definable time cost, complexity of algorithms became a field of study; since computing was no longer performed in 'real' numbers but in representations in finite bit-strings, the accuracy of algorithms needed to be studied. Some of these considerations in fact predate the existence of computers, having been inspired by computing with mechanical calculators.

A prime concern in scientific computing is efficiency. While to some scientists the abstract fact of the existence of a solution is enough, in computing we actually want that solution, and preferably yesterday. For this reason, in this book we will be quite specific about the efficiency of both algorithms and hardware. It is important not to limit the concept of efficiency to that of efficient use of hardware. While this is important, the difference between two algorithmic approaches can make optimization for specific hardware a secondary concern.

This book aims to cover the basics of this gamut of knowledge that a successful computational scientist needs to master. It is set up as a textbook for graduate students or advanced undergraduate students; others can use it as a reference text, reading the exercises for their information content.

Contents

I Theory 11
1 **Single-processor Computing** 12
1.1 *The von Neumann architecture* 12
1.2 *Modern processors* 14
1.3 *Memory Hierarchies* 21
1.4 *Multicore architectures* 38
1.5 *Node architecture and sockets* 44
1.6 *Locality and data reuse* 46
1.7 *Further topics* 53
1.8 *Review questions* 56
2 **Parallel Computing** 57
2.1 *Introduction* 57
2.2 *Theoretical concepts* 61
2.3 *Parallel Computers Architectures* 74
2.4 *Different types of memory access* 78
2.5 *Granularity of parallelism* 81
2.6 *Parallel programming* 85
2.7 *Topologies* 120
2.8 *Multi-threaded architectures* 134
2.9 *Co-processors, including GPUs* 135
2.10 *Load balancing* 140
2.11 *Remaining topics* 147
3 **Computer Arithmetic** 157
3.1 *Bits* 157
3.2 *Integers* 158
3.3 *Real numbers* 162
3.4 *The IEEE 754 standard for floating point numbers* 167
3.5 *Round-off error analysis* 171
3.6 *Examples of round-off error* 174
3.7 *Computer arithmetic in programming languages* 180
3.8 *More about floating point arithmetic* 186
3.9 *Conclusions* 190
3.10 *Review questions* 191

4 **Numerical treatment of differential equations** *192*
4.1 *Initial value problems* *192*
4.2 *Boundary value problems* *199*
4.3 *Initial boundary value problem* *207*
5 **Numerical linear algebra** *212*
5.1 *Elimination of unknowns* *212*
5.2 *Linear algebra in computer arithmetic* *215*
5.3 *LU factorization* *217*
5.4 *Sparse matrices* *227*
5.5 *Iterative methods* *241*
5.6 *Eigenvalue methods* *260*
5.7 *Further Reading* *260*
6 **Programming for performance** *261*
6.1 *Peak performance* *261*
6.2 *Bandwidth* *261*
6.3 *Arithmetic performance* *264*
6.4 *Hardware exploration* *267*
6.5 *Cache associativity* *271*
6.6 *Loop nests* *273*
6.7 *Loop tiling* *274*
6.8 *Optimization strategies* *276*
6.9 *Cache aware and cache oblivious programming* *277*
6.10 *Case study: Matrix-vector product* *278*
7 **High performance linear algebra** *281*
7.1 *Collective operations* *281*
7.2 *Parallel dense matrix-vector product* *285*
7.3 *LU factorization in parallel* *295*
7.4 *Matrix-matrix product* *300*
7.5 *Sparse matrix-vector product* *303*
7.6 *Computational aspects of iterative methods* *311*
7.7 *Parallel preconditioners* *315*
7.8 *Ordering strategies and parallelism* *317*
7.9 *Parallelism in solving linear systems from Partial Diffential Equations (PDEs)* *329*
7.10 *Parallelism and implicit operations* *331*
7.11 *Grid updates* *337*
7.12 *Block algorithms on multicore architectures* *339*

II Applications *345*
8 **Molecular dynamics** *346*
8.1 *Force Computation* *347*
8.2 *Parallel Decompositions* *350*
8.3 *Parallel Fast Fourier Transform* *357*

8.4 *Integration for Molecular Dynamics* 359
9 **Combinatorial algorithms** 363
9.1 *Brief introduction to sorting* 363
9.2 *Odd-even transposition sort* 365
9.3 *Quicksort* 366
9.4 *Radixsort* 368
9.5 *Samplesort* 370
9.6 *Bitonic sort* 371
9.7 *Prime number finding* 373
10 **Graph analytics** 374
10.1 *Traditional graph algorithms* 374
10.2 *Linear algebra on the adjacency matrix* 379
10.3 *'Real world' graphs* 383
10.4 *Hypertext algorithms* 384
10.5 *Large-scale computational graph theory* 386
11 **N-body problems** 388
11.1 *The Barnes-Hut algorithm* 389
11.2 *The Fast Multipole Method* 390
11.3 *Full computation* 390
11.4 *Implementation* 391
12 **Monte Carlo Methods** 396
12.1 *Motivation* 396
12.2 *Examples* 398
13 **Machine learning** 400
13.1 *Neural networks* 400
13.2 *Deep learning networks* 402
13.3 *Computational aspects* 405
13.4 *Stuff* 408

III Appendices 409
14 **Linear algebra** 411
14.1 *Norms* 411
14.2 *Gram-Schmidt orthogonalization* 412
14.3 *The power method* 414
14.4 *Nonnegative matrices; Perron vectors* 416
14.5 *The Gershgorin theorem* 416
14.6 *Householder reflectors* 417
15 **Complexity** 418
15.1 *Formal definitions* 418
15.2 *The Master Theorem* 419
15.3 *Little-oh complexity* 420
15.4 *Amortized complexity* 421

16 **Partial Differential Equations** *422*
16.1 *Partial derivatives* *422*
16.2 *Poisson or Laplace Equation* *422*
16.3 *Heat Equation* *423*
16.4 *Steady state* *423*
17 **Taylor series** *424*
18 **Minimization** *427*
18.1 *Descent methods* *427*
18.2 *Newton's method* *429*
19 **Random numbers** *432*
19.1 *Random Number Generation* *432*
19.2 *Random numbers in programming languages* *433*
19.3 *Parallel random number generation* *436*
20 **Graph theory** *438*
20.1 *Definitions* *438*
20.2 *Common types of graphs* *439*
20.3 *Graph colouring and independent sets* *440*
20.4 *Graph algorithms* *441*
20.5 *Graphs and matrices* *441*
20.6 *Spectral graph theory* *443*
21 **Automata theory** *446*
21.1 *Finite State Automata* *446*
21.2 *General discussion* *446*
22 **Parallel Prefix** *448*
22.1 *Parallel prefix* *448*
22.2 *Sparse matrix vector product as parallel prefix* *449*
22.3 *Horner's rule* *450*

IV Projects, codes *453*
23 **Class projects** *454*
24 **Teaching guide** *455*
24.1 *Standalone course* *455*
24.2 *Addendum to parallel programming class* *455*
24.3 *Tutorials* *455*
25 **Cache simulation and analysis** *456*
25.1 *Cache simulation* *456*
25.2 *Code simulation* *456*
25.3 *Investigation* *457*
25.4 *Analysis* *457*
26 **Evaluation of Bulk Synchronous Programming** *458*
26.1 *Discussion* *458*
26.2 *Simulation* *458*

27 **Heat equation** *459*
27.1 *Software* *459*
27.2 *Tests* *460*
27.3 *Parallelism* *461*
27.4 *Comparison of solvers* *461*
27.5 *Reporting* *462*
28 **The memory wall** *463*
28.1 *Background* *463*
28.2 *Assignment* *463*
29 **Codes** *464*
29.1 *Preliminaries* *464*
29.2 *Cache size* *465*
29.3 *Cachelines* *467*
29.4 *Cache associativity* *469*
29.5 *TLB* *470*

V Indices *473*
30 **Index** *474*
Index of concepts and names *474*
31 **List of acronyms** *475*
32 **Bibliography** *476*

PART I

THEORY

Chapter 1

Single-processor Computing

In order to write efficient scientific codes, it is important to understand computer architecture. The difference in speed between two codes that compute the same result can range from a few percent to orders of magnitude, depending only on factors relating to how well the algorithms are coded for the processor architecture. Clearly, it is not enough to have an algorithm and 'put it on the computer': some knowledge of computer architecture is advisable, sometimes crucial.

Some problems can be solved on a single Central Processing Unit (CPU), others need a parallel computer that comprises more than one processor. We will go into detail on parallel computers in the next chapter, but even for parallel processing, it is necessary to understand the individual CPUs.

In this chapter, we will focus on what goes on inside a CPU and its memory system. We start with a brief general discussion of how instructions are handled, then we will look into the arithmetic processing in the processor core; last but not least, we will devote much attention to the movement of data between memory and the processor, and inside the processor. This latter point is, maybe unexpectedly, very important, since memory access is typically much slower than executing the processor's instructions, making it the determining factor in a program's performance; the days when 'flops (Floating Point Operations per Second) counting' was the key to predicting a code's performance are long gone. This discrepancy is in fact a growing trend, so the issue of dealing with memory traffic has been becoming more important over time, rather than going away.

This chapter will give you a basic understanding of the issues involved in CPU design, how it affects performance, and how you can code for optimal performance. For much more detail, see the standard work about computer architecture, Hennesey and Patterson [100].

1.1 The von Neumann architecture

While computers, and most relevantly for this chapter, their processors, can differ in any number of details, they also have many aspects in common. On a very high level of abstraction, many architectures can be described as *von Neumann architectures*. This describes a design with an undivided memory that stores both program and data ('stored program'), and a processing unit that executes the instructions, operating on the data in 'fetch, execute, store cycle'.

Remark *This model with a prescribed sequence of instructions is also referred to as* control flow. *This is in contrast to* data flow*, which we will see in section 7.12.*

This setup distinguishes modern processors for the very earliest, and some special purpose contemporary, designs where the program was hard-wired. It also allows programs to modify themselves or generate other programs, since instructions and data are in the same storage. This allows us to have editors and compilers: the computer treats program code as data to operate on. **Remark** *At one time, the stored program concept was included as essential for the ability for a running program to modify its own source. However, it was quickly recognized that this leads to unmaintainable code, and is rarely done in practice [49].* In this book we will not explicitly discuss compilation, the process that translates high level languages to machine instructions. (See the tutorial *Tutorials book, section 2* for usage aspects.) However, on occasion we will discuss how a program at high level can be written to ensure efficiency at the low level.

In scientific computing, however, we typically do not pay much attention to program code, focusing almost exclusively on data and how it is moved about during program execution. For most practical purposes it is as if program and data are stored separately. The little that is essential about instruction handling can be described as follows.

The machine instructions that a processor executes, as opposed to the higher level languages users write in, typically specify the name of an operation, as well as of the locations of the operands and the result. These locations are not expressed as memory locations, but as *registers*: a small number of named memory locations that are part of the CPU. **Remark** *Direct-to-memory architectures are rare, though they have existed. The Cyber 205 supercomputer in the 1980s could have three data streams, two from memory to the processor, and one back from the processor to memory, going on at the same time. Such an architecture is only feasible if memory can keep up with the processor speed, which is no longer the case these days.* As an example, here is a simple C routine

```
void store(double *a, double *b, double *c) {
 *c = *a + *b;
}
```

and its X86 assembler output, obtained by `gcc -O2 -S -o - store.c`:

```
        .text
        .p2align 4,,15
.globl store
        .type   store, @function
store:
        movsd   (%rdi), %xmm0           # Load *a to %xmm0
        addsd   (%rsi), %xmm0           # Load *b and add to %xmm0
        movsd   %xmm0, (%rdx)           # Store to *c
        ret
```

(This is 64-bit output; add the option `-m64` on 32-bit systems.)

The instructions here are:

- A load from memory to register;
- Another load, combined with an addition;
- Writing back the result to memory.

Each instruction is processed as follows:

- Instruction fetch: the next instruction according to the *program counter* is loaded into the processor. We will ignore the questions of how and from where this happens.
- Instruction decode: the processor inspects the instruction to determine the operation and the operands.
- Memory fetch: if necessary, data is brought from memory into a register.
- Execution: the operation is executed, reading data from registers and writing it back to a register.
- Write-back: for store operations, the register contents is written back to memory.

The case of array data is a little more complicated: the element loaded (or stored) is then determined as the base address of the array plus an offset.

In a way, then, the modern CPU looks to the programmer like a von Neumann machine. There are various ways in which this is not so. For one, while memory looks randomly addressable[1], in practice there is a concept of *locality*: once a data item has been loaded, nearby items are more efficient to load, and reloading the initial item is also faster.

Another complication to this story of simple loading of data is that contemporary CPUs operate on several instructions simultaneously, which are said to be 'in flight', meaning that they are in various stages of completion. Of course, together with these simultaneous instructions, their inputs and outputs are also being moved between memory and processor in an overlapping manner. This is the basic idea of the *superscalar* CPU architecture, and is also referred to as *Instruction Level Parallelism (ILP)*. Thus, while each instruction can take several clock cycles to complete, a processor can complete one instruction per cycle in favorable circumstances; in some cases more than one instruction can be finished per cycle.

The main statistic that is quoted about CPUs is their Gigahertz rating, implying that the speed of the processor is the main determining factor of a computer's performance. While speed obviously correlates with performance, the story is more complicated. Some algorithms are *cpu-bound*, and the speed of the processor is indeed the most important factor; other algorithms are *memory-bound*, and aspects such as bus speed and cache size, to be discussed later, become important.

In scientific computing, this second category is in fact quite prominent, so in this chapter we will devote plenty of attention to the process that moves data from memory to the processor, and we will devote relatively little attention to the actual processor.

1.2 Modern processors

Modern processors are quite complicated, and in this section we will give a short tour of what their constituent parts. Figure 1.1 is a picture of the *die* of an *Intel Sandy Bridge* processor. This chip is about an inch in size and contains close to a billion transistors.

1. There is in fact a theoretical model for computation called the 'Random Access Machine'; we will briefly see its parallel generalization in section 2.2.2.

1.2.1 The processing cores

In the Von Neumann model there is a single entity that executes instructions. This has not been the case in increasing measure since the early 2000s. The Sandy Bridge pictured in figure 1.1 has eight *cores*, each of which is an independent unit executing a stream of instructions. In this chapter we will mostly discuss aspects of a single *core*; section 1.4 will discuss the integration aspects of the multiple cores.

1.2.1.1 Instruction handling

The *von Neumann architecture* model is also unrealistic in that it assumes that all instructions are executed strictly in sequence. Increasingly, over the last twenty years, processor have used *out-of-order* instruction handling, where instructions can be processed in a different order than the user program specifies. Of course the processor is only allowed to re-order instructions if that leaves the result of the execution intact!

In the block diagram (figure 1.2) you see various units that are concerned with instruction handling: This cleverness actually costs considerable energy, as well as sheer amount of transistors. For this reason, processors such as the first generation Intel Xeon Phi, the *Knights Corner*, used *in-order* instruction handling. However, in the next generation, the *Knights Landing*, this decision was reversed for reasons of performance.

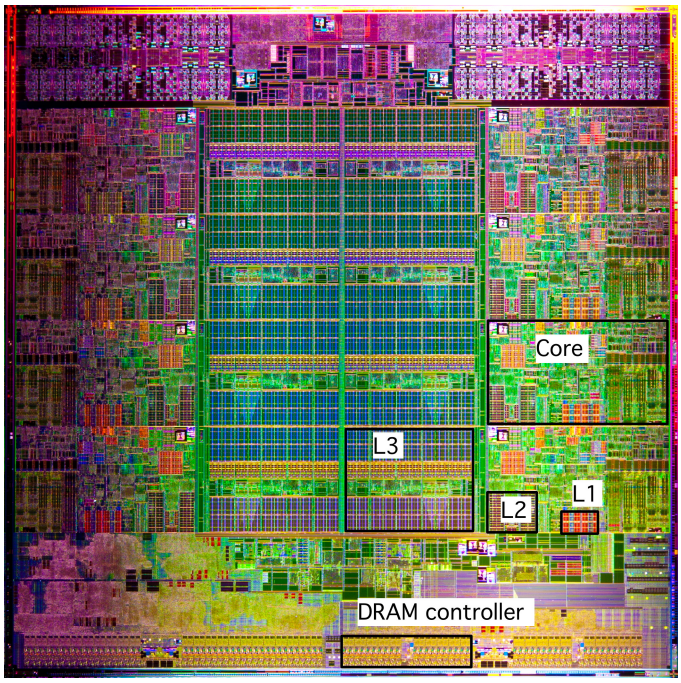


Figure 1.1: The Intel Sandy Bridge processor die.

1.2.1.2 Floating point units

In scientific computing we are mostly interested in what a processor does with floating point data. Computing with integers or booleans is typically of less interest. For this reason, cores have considerable sophistication for dealing with numerical data.

For instance, while past processors had just a single Floating Point Unit (FPU), these days they will have multiple, capable of executing simultaneously.

For instance, often there are separate addition and multiplication units; if the compiler can find addition and multiplication operations that are independent, it can schedule them so as to be executed simultaneously, thereby doubling the performance of the processor. In some cases, a processor will have multiple addition or multiplication units.

Another way to increase performance is to have a *Fused Multiply-Add (FMA)* unit, which can execute the instruction $x \leftarrow ax + b$ in the same amount of time as a separate addition or multiplication. Together with pipelining (see below), this means that a processor has an asymptotic speed of several floating point operations per clock cycle.

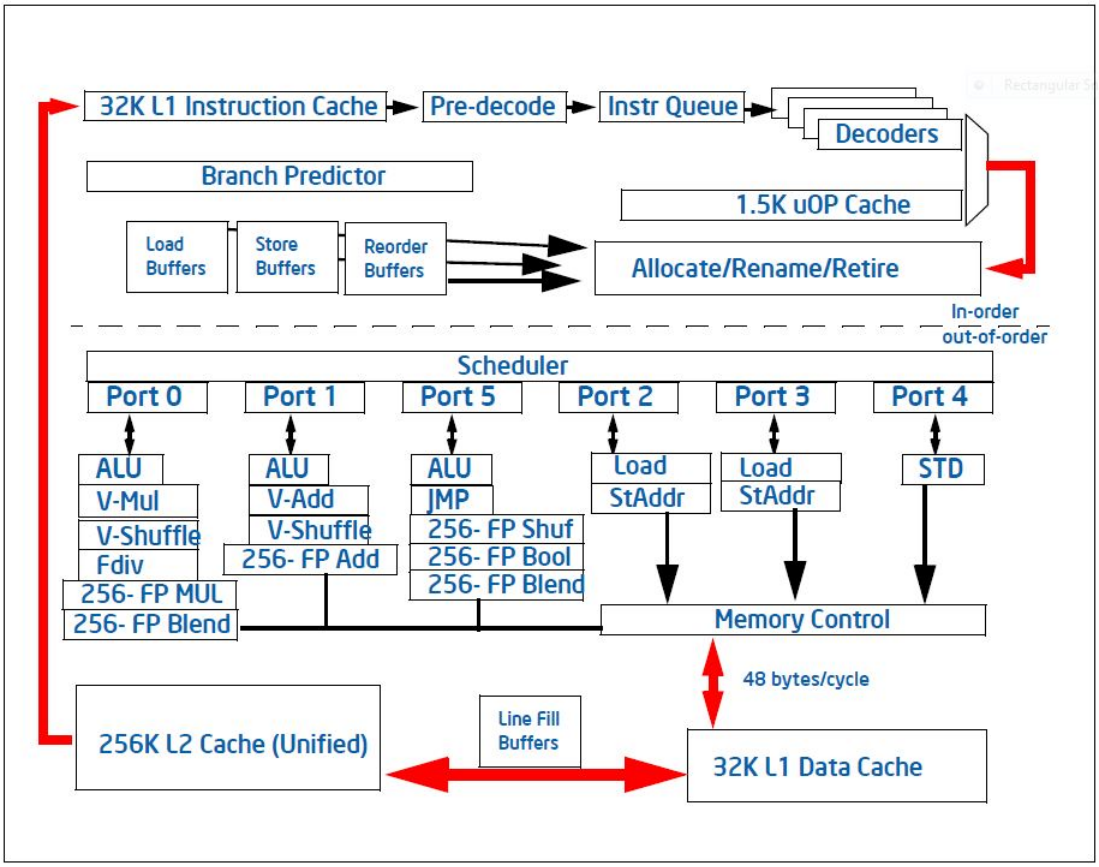


Figure 2-1. Intel microarchitecture code name Sandy Bridge Pipeline Functionality

Figure 1.2: Block diagram of the Intel Sandy Bridge core.

Incidentally, there are few algorithms in which division operations are a limiting factor. Correspondingly, the division operation is not nearly as much optimized in a modern CPU as the additions and multiplications are. Division operations can take 10 or 20 clock cycles, while a CPU can have multiple addition and/or multiplication units that (asymptotically) can produce a result per cycle.

1.2.1.3 Pipelining

The floating point add and multiply units of a processor are pipelined, which has the effect that a stream of independent operations can be performed at an asymptotic speed of one result per clock cycle.

The idea behind a pipeline is as follows. Assume that an operation consists of multiple simpler operations, then we can potentially speed up the operation by using dedicated hardware for each suboperation. If we

Processor	year	add/mult/fma units (count×width)	daxpy cycles (arith vs load/store)
MIPS R10000	1996	$1\times1+1\times1+0$	8/24
Alpha EV5	1996	$1\times1+1\times1+0$	8/12
IBM Power5	2004	$0+0+2\times1$	4/12
AMD Bulldozer	2011	$2\times2+2\times2+0$	2/4
Intel Sandy Bridge	2012	$1\times4+1\times4+0$	2/4
Intel Haswell	2014	$0+0+2\times4$	1/2

Table 1.1: Floating point capabilities (per core) of several processor architectures, and DAXPY cycle number for 8 operands.

now have multiple operations to perform, we get a speedup by having all suboperations active simultaneously: each hands its result to the next and accepts its input(s) from the previous.

For instance, an addition instruction can have the following components:

- Decoding the instruction, including finding the locations of the operands.
- Copying the operands into registers ('data fetch').
- Aligning the exponents; the addition $.35\times10^{-1}+.6\times10^{-2}$ becomes $.35\times10^{-1}+.06\times10^{-1}$.
- Executing the addition of the mantissas, in this case giving .41.
- Normalizing the result, in this example to $.41\times10^{-1}$. (Normalization in this example does not do anything. Check for yourself that in $.3\times10^{0}+.8\times10^{0}$ and $.35\times10^{-3}+(-.34)\times10^{-3}$ there is a non-trivial adjustment.)
- Storing the result.

These parts are often called the 'stages' or 'segments' of the pipeline.

If every component is designed to finish in 1 clock cycle, the whole instruction takes 6 cycles. However, if each has its own hardware, we can execute two operations in less than 12 cycles:

- Execute the decode stage for the first operation;
- Do the data fetch for the first operation, and at the same time the decode for the second.
- Execute the third stage for the first operation and the second stage of the second operation simultaneously.
- Et cetera.

You see that the first operation still takes 6 clock cycles, but the second one is finished a mere 1 cycle later.

Let us make a formal analysis of the speedup you can get from a pipeline. On a traditional FPU, producing n results takes $t(n)=n\ell\tau$ where ℓ is the number of stages, and τ the clock cycle time. The rate at which results are produced is the reciprocal of $t(n)/n$: $r_{\text{serial}}\equiv(\ell\tau)^{-1}$.

On the other hand, for a pipelined FPU the time is $t(n)=[s+\ell+n-1]\tau$ where s is a setup cost: the first operation still has to go through the same stages as before, but after that one more result will be produced each cycle. We can also write this formula as

$$t(n)=[n+n_{1/2}]\tau,$$

expressing the linear time, plus an offset.

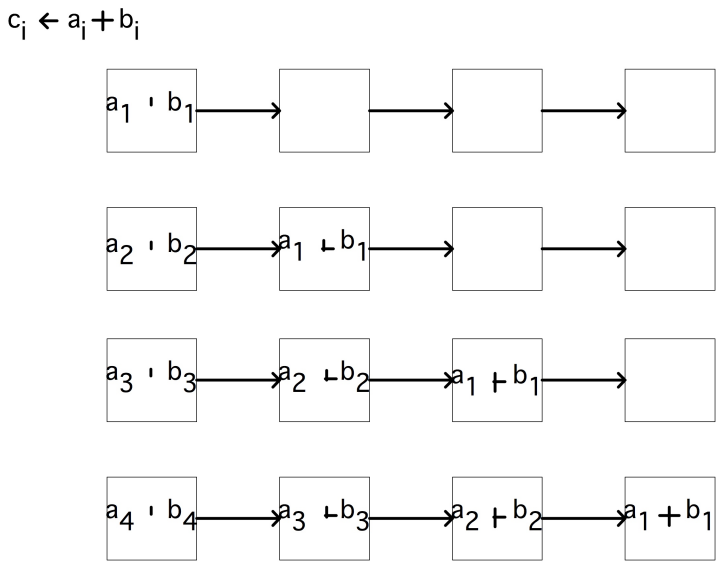


Figure 1.3: Schematic depiction of a pipelined operation.

Exercise 1.1. Let us compare the speed of a classical FPU, and a pipelined one. Show that the result rate is now dependent on n: give a formula for $r(n)$, and for $r_\infty = \lim_{n\to\infty} r(n)$. What is the asymptotic improvement in r over the non-pipelined case?
Next you can wonder how long it takes to get close to the asymptotic behavior. Show that for $n = n_{1/2}$ you get $r(n) = r_\infty/2$. This is often used as the definition of $n_{1/2}$.

Since a vector processor works on a number of instructions simultaneously, these instructions have to be independent. The operation $\forall_i : a_i \leftarrow b_i + c_i$ has independent additions; the operation $\forall_i : a_{i+1} \leftarrow a_i b_i + c_i$ feeds the result of one iteration (a_i) to the input of the next ($a_{i+1} = \ldots$), so the operations are not independent.

A pipelined processor can speed up operations by a factor of $4, 5, 6$ with respect to earlier CPUs. Such numbers were typical in the 1980s when the first successful vector computers came on the market. These days, CPUs can have 20-stage pipelines. Does that mean they are incredibly fast? This question is a bit complicated. Chip designers continue to increase the clock rate, and the pipeline segments can no longer finish their work in one cycle, so they are further split up. Sometimes there are even segments in which nothing happens: that time is needed to make sure data can travel to a different part of the chip in time.

The amount of improvement you can get from a pipelined CPU is limited, so in a quest for ever higher performance several variations on the pipeline design have been tried. For instance, the Cyber 205 had separate addition and multiplication pipelines, and it was possible to feed one pipe into the next without data going back to memory first. Operations like $\forall_i : a_i \leftarrow b_i + c \cdot d_i$ were called 'linked triads' (because of the number of paths to memory, one input operand had to be scalar).

Exercise 1.2. Analyze the speedup and $n_{1/2}$ of linked triads.

Another way to increase performance is to have multiple identical pipes. This design was perfected by the NEC SX series. With, for instance, 4 pipes, the operation $\forall_i : a_i \leftarrow b_i + c_i$ would be split module 4, so that the first pipe operated on indices $i = 4 \cdot j$, the second on $i = 4 \cdot j + 1$, et cetera.

Exercise 1.3. Analyze the speedup and $n_{1/2}$ of a processor with multiple pipelines that operate in parallel. That is, suppose that there are p independent pipelines, executing the same instruction, that can each handle a stream of operands.

(You may wonder why we are mentioning some fairly old computers here: true pipeline supercomputers hardly exist anymore. In the US, the Cray X1 was the last of that line, and in Japan only NEC still makes them. However, the functional units of a CPU these days are pipelined, so the notion is still important.)

Exercise 1.4. The operation

```
for (int i=0; i<size; ++i) {
  x[i+1] = a[i]*x[i] + b[i];
}
```

can not be handled by a pipeline because there is a *dependency* between input of one iteration of the operation and the output of the previous. However, you can transform the loop into one that is mathematically equivalent, and potentially more efficient to compute. Derive an expression that computes `x[i+2]` from `x[i]` without involving `x[i+1]`. This is known as *recursive doubling*. Assume you have plenty of temporary storage. You can now perform the calculation by

- Doing some preliminary calculations;
- computing `x[i]`, `x[i+2]`, `x[i+4]`, ..., and from these,
- compute the missing terms `x[i+1]`, `x[i+3]`, ...

Analyze the efficiency of this scheme by giving formulas for $T_0(n)$ and $T_s(n)$. Can you think of an argument why the preliminary calculations may be of lesser importance in some circumstances?

1.2.1.3.1 **Systolic computing** Pipelining as described above is one case of a *systolic algorithm*. In the 1980s and 1990s there was research into using pipelined algorithms and building special hardware, *systolic array*s, to implement them [128]. This is also connected to computing with Field Programmable Gate Arrays (FPGAs), where the systolic array is software-defined.

Section 6.3.2 does a performance study of pipelining and loop unrolling.

1.2.1.4 Peak performance

Thanks to pipelining, for modern CPUs there is a simple relation between the *clock speed* and the *peak performance*. Since each FPU can produce one result per cycle asymptotically, the peak performance is the clock speed times the number of independent FPUs. The measure of floating point performance is 'floating point operations per second', abbreviated *flops*. Considering the speed of computers these days, you will mostly hear floating point performance being expressed in 'gigaflops': multiples of 10^9 flops.

1.2.2 8-bit, 16-bit, 32-bit, 64-bit

Processors are often characterized in terms of how big a chunk of data they can process as a unit. This can relate to

- The width of the path between processor and memory: can a 64-bit floating point number be loaded in one cycle, or does it arrive in pieces at the processor.
- The way memory is addressed: if addresses are limited to 16 bits, only 64,000 bytes can be identified. Early PCs had a complicated scheme with segments to get around this limitation: an address was specified with a segment number and an offset inside the segment.
- The number of bits in a register, in particular the size of the integer registers which manipulate data address; see the previous point. (Floating point register are often larger, for instance 80 bits in the x86 architecture.) This also corresponds to the size of a chunk of data that a processor can operate on simultaneously.
- The size of a floating point number. If the arithmetic unit of a CPU is designed to multiply 8-byte numbers efficiently ('double precision'; see section 3.3.2) then numbers half that size ('single precision') can sometimes be processed at higher efficiency, and for larger numbers ('quadruple precision') some complicated scheme is needed. For instance, a quad precision number could be emulated by two double precision numbers with a fixed difference between the exponents.

These measurements are not necessarily identical. For instance, the original Pentium processor had 64-bit data busses, but a 32-bit processor. On the other hand, the Motorola 68000 processor (of the original Apple Macintosh) had a 32-bit CPU, but 16-bit data busses.

The first Intel microprocessor, the 4004, was a 4-bit processor in the sense that it processed 4 bit chunks. These days, 64 bit processors are becoming the norm.

1.2.3 Caches: on-chip memory

The bulk of computer memory is in chips that are separate from the processor. However, there is usually a small amount (typically a few megabytes) of on-chip memory, called the *cache*. This will be explained in detail in section 1.3.5.

1.2.4 Graphics, controllers, special purpose hardware

One difference between 'consumer' and 'server' type processors is that the consumer chips devote considerable real-estate on the processor chip to graphics. Processors for cell phones and tablets can even have dedicated circuitry for security or mp3 playback. Other parts of the processor are dedicated to communicating with memory or the *I/O subsystem*. We will not discuss those aspects in this book.

1.2.5 Superscalar processing and instruction-level parallelism

In the *von Neumann architecture* model, processors operate through *control flow*: instructions follow each other linearly or with branches without regard for what data they involve. As processors became more powerful and capable of executing more than one instruction at a time, it became necessary to switch to

the *data flow* model. Such *superscalar* processors analyze several instructions to find data dependencies, and execute instructions in parallel that do not depend on each other.

This concept is also known as *Instruction Level Parallelism (ILP)*, and it is facilitated by various mechanisms:

- multiple-issue: instructions that are independent can be started at the same time;
- pipelining: arithmetic units can deal with multiple operations in various stages of completion (section 1.2.1.3);
- branch prediction and speculative execution: a compiler can 'guess' whether a conditional instruction will evaluate to true, and execute those instructions accordingly;
- out-of-order execution: instructions can be rearranged if they are not dependent on each other, and if the resulting execution will be more efficient;
- *prefetching*: data can be speculatively requested before any instruction needing it is actually encountered (this is discussed further in section 1.3.6).

Above, you saw pipelining in the context of floating point operations. Nowadays, the whole CPU is pipelined. Not only floating point operations, but any sort of instruction will be put in the *instruction pipeline* as soon as possible. Note that this pipeline is no longer limited to identical instructions: the notion of pipeline is now generalized to any stream of partially executed instructions that are simultaneously "in flight".

As clock frequency has gone up, the processor pipeline has grown in length to make the segments executable in less time. You have already seen that longer pipelines have a larger $n_{1/2}$, so more independent instructions are needed to make the pipeline run at full efficiency. As the limits to instruction-level parallelism are reached, making pipelines longer (sometimes called 'deeper') no longer pays off. This is generally seen as the reason that chip designers have moved to *multicore* architectures as a way of more efficiently using the transistors on a chip; see section 1.4.

There is a second problem with these longer pipelines: if the code comes to a branch point (a conditional or the test in a loop), it is not clear what the next instruction to execute is. At that point the pipeline can *stall*. CPUs have taken to *speculative execution* for instance, by always assuming that the test will turn out true. If the code then takes the other branch (this is called a *branch misprediction*), the pipeline has to be *flushed* and restarted. The resulting delay in the execution stream is called the *branch penalty*.

1.3 Memory Hierarchies

We will now refine the picture of the *von Neumann architecture* from section 1.1.

To the programmer, a computer appears to execute a sequence of steps:

1. decode an instruction to determine the operands,
2. retrieve the operands from memory,
3. execute the operation and write the result back to memory.

This picture, known as the *von Neumann architecture*, is unrealistic because of the so-called *memory wall* [198], also known as the *von Neumann bottleneck*: memory is too slow to load data into the processor at the rate the processor can absorb it. Specifically, a single load can have a latency of hundreds

of cycles, meaning that this much time elapses between a processor requesting a memory item and it actually arriving for processing. On the other hand, a processor can perform several operations per cycle. After this long wait for a load, the next load can come faster, but you may still be limited by the *bandwidth* of the memory, which is lower than what the processor can operate on.

To circumvent this imbalance, a modern CPU has several memory levels in between the FPU and the main memory: the registers and the caches, together called the *memory hierarchy*. These try to alleviate the memory wall problem by making recently used data available quicker than it would be from main memory.

Of course, this presupposes that the algorithm and its implementation allow for data to be used multiple times. Such questions of *data reuse* will be discussed in more detail in section 1.6.1.

Both registers and caches are faster than main memory to various degrees; unfortunately, the faster the memory on a certain level, the smaller it will be. These differences in size and access speed lead to interesting programming problems, which we will discuss later. See chapter 6.

We will now discuss the various components of the memory hierarchy and the theoretical concepts needed to analyze their behavior.

1.3.1 Busses

The wires that move data around in a computer, from memory to cpu or to a disc controller or screen, are called *busses*. The most important one for us is the *Front-Side Bus (FSB)* which connects the processor to memory. In one popular architecture, this is called the 'north bridge', as opposed to the 'south bridge' which connects to external devices, with the exception of the graphics controller.

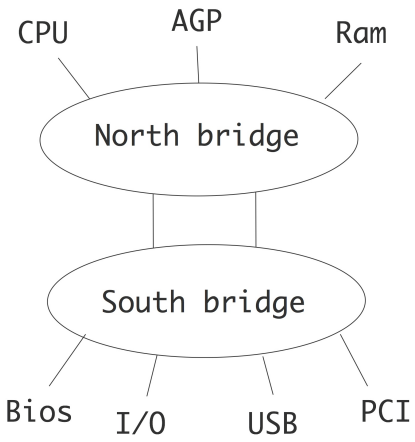


Figure 1.4: Bus structure of a processor.

The bus is typically slower than the processor, operating with clock frequencies slightly in excess of 1GHz, which is a fraction of the CPU clock frequency. This is one reason that caches are needed; the fact that a processor can consume many data items per clock tick contributes to this. Apart from the frequency, the

bandwidth of a bus is also determined by the number of bits that can be moved per clock cycle. This is typically 64 or 128 in current architectures. We will now discuss this in some more detail.

1.3.2 Latency and Bandwidth

Above, we mentioned in very general terms that accessing data in registers is almost instantaneous, whereas loading data from memory into the registers, a necessary step before any operation, incurs a substantial delay. We will now make this story slightly more precise.

There are two important concepts to describe the movement of data: *latency* and *bandwidth*. The assumption here is that requesting an item of data incurs an initial delay; if this item was the first in a stream of data, usually a consecutive range of memory addresses, the remainder of the stream will arrive with no further delay at a regular amount per time period.

Latency is the delay between the processor issuing a request for a memory item, and the item actually arriving. We can distinguish between various latencies, such as the transfer from memory to cache, cache to register, or summarize them all into the latency between memory and processor. Latency is measured in (nano) seconds, or clock periods.
If a processor executes instructions in the order they are found in the assembly code, then execution will often *stall* while data is being fetched from memory; this is also called *memory stall*. For this reason, a low latency is very important. In practice, many processors have 'out-of-order execution' of instructions, allowing them to perform other operations while waiting for the requested data. Programmers can take this into account, and code in a way that achieves *latency hiding*; see also section 1.6.1. Graphics Processing Units (GPUs) (see section 2.9.3) can switch very quickly between threads in order to achieve latency hiding.

Bandwidth is the rate at which data arrives at its destination, after the initial latency is overcome. Bandwidth is measured in bytes (kilobytes, megabytes, gigabytes) per second or per clock cycle. The bandwidth between two memory levels is usually the product of:

- the cycle speed of the channel (the *bus speed*), measured in GT/s),
- the *bus width*: the number of bits that can be sent simultaneously in every cycle of the bus clock.
- the number of channels per socket,
- the number of sockets.

For example, for the *TACC Frontera* cluster, the computation is

$$\text{bandwidth} = 2.933\,GT/s \times 8\,\text{Bytes/Transfer/channel} \times 6\text{channels/socket} \times 2\textit{sockets}.$$

The concepts of latency and bandwidth are often combined in a formula for the time that a message takes from start to finish:

$$T(n) = \alpha + \beta n$$

where α is the latency and β is the inverse of the bandwidth: the time per byte.

The further away from the processor one gets in the memory hierarchy, the longer the latency is, and the lower the bandwidth. This makes it important to program in such a way that, if at all possible, the processor uses data from cache or register, rather than from main memory.

1.3.3 Streaming operations

The worst case, performancewise speaking, of an operation is a so-called 'streaming' kernel. Here, data streams from and to memory, with effectively no use of caches to increase performance. As an example, consider the vector addition:

```
for (int i=0; i<data_size; ++i)
  a[i] = b[i]+c[i]
```

Each iteration performs one floating point operation, which modern CPUs can do in one clock cycle by using pipelines. However, each iteration needs two numbers loaded and one written for a total of 24 bytes of memory traffic. (Actually, the cacheline of `a[i]` is loaded before it can be written, so there are 4 memory accesses with a total of 32 bytes per iteration.) Typical memory bandwidth figures (see for instance figure 1.5) are nowhere near 24 (or 32) bytes per cycle. This means that, without caches, algorithm performance can be bounded by memory performance. Of course, caches will not speed up every operation, and in fact will have no effect on the above example. Strategies for programming that lead to significant cache use are discussed in chapter 6.

The concepts of latency and bandwidth will also appear in parallel computers, when we talk about sending data from one processor to the next.

1.3.4 Registers

Every processor has a small amount of memory that is internal to the processor: the *registers*, or together the *register file*. The registers are what the processor actually operates on: an operation such as

```
a = b + c
```

can be thought of as being implemented as

- load the value of `b` from memory into a register,
- load the value of `c` from memory into another register,
- compute the sum and write that into the same or yet another register, and
- write the sum value back to the memory location of `a`.

Looking at *assembly code* (for instance the output of a compiler; section **??**) you see explicit load, compute, and store instructions, such as

```
addl    %eax, %edx
```

which adds the content of one register to another. As you see in this sample instruction, registers are not numbered, as opposed to memory addresses, but have distinct names that are referred to in the assembly instruction. Typically, a processor has 16 or 32 floating point registers; the *Intel Itanium* was exceptional with 128 floating point registers.

Registers have a high bandwidth and low latency because they are part of the processor. You can consider data movement to and from registers as essentially instantaneous.

In this chapter you will see stressed that moving data from memory is relatively expensive. Therefore, it would be a simple optimization to leave data in register when possible.

Registers are a first way of minimizing expensive data movement. For instance, a strict interpretation of these instructions:

```
a = b + c
d = a + e
```

would see `a` written to main memory after the first instruction, and the reloaded for the second. As an obvious optimization, the computed value of `a` could be left in register: we say that `a` stays *resident in register*. This is typically performed as a *compiler optimization*.

Keeping values in register is often done to avoid recomputing a quantity. For instance, in

```
t1 = sin(alpha) * x + cos(alpha) * y;
t2 = -cos(alpha) * x + sin(alpha) * y;
```

the sine and cosine quantity will probably be kept in register. You can help the compiler by explicitly introducing temporary quantities:

```
s = sin(alpha); c = cos(alpha);
t1 = s * x + c * y;
t2 = -c * x + s * y
```

However, compilers can often figure out such transformations by themselves, if you set the *optimization level* high enough; see *Tutorials book, section 2.2.4*.

Of course, there is a limit to how many quantities can be kept in register; trying to keep too many quantities in register is called *register spill* and lowers the performance of a code.

Keeping a variable in register is especially important if that variable appears in an inner loop. In the computation

```
for i=1,length
  a[i] = b[i] * c
```

the quantity `c` will probably be kept in register by the compiler, but in

```
for k=1,nvectors
  for i=1,length
    a[i,k] = b[i,k] * c[k]
```

it is a good idea to introduce explicitly a temporary variable to hold `c[k]`. In C, you can give a hint to the compiler to keep a variable in register by declaring it as a *register variable*:

```
register double t;
```

However, compilers are clever enough these days about register allocation that such hints are likely to be ignored.

1.3.5 Caches

In between the registers, which contain the immediate input and output data for instructions, and the main memory where lots of data can reside for a long time, are various levels of *cache* memory, that have

lower latency and higher bandwidth than main memory and where data are kept for an intermediate amount of time.

Data from memory travels through the caches to wind up in registers. The advantage to having cache memory is that if a data item is reused shortly after it was first needed, it will still be in cache, and therefore it can be accessed much faster than if it had to be brought in from memory.

This section discusses the idea behind caches; for cache-aware programming, see section 6.4.1.

On a historical note, the notion of levels of memory hierarchy was already discussed in 1946 [25], motivated by the slowness of the memory technology at the time.

1.3.5.1 A motivating example

To motivate the use of caches let's suppose a variable x is used twice, and that it does not stay resident in a register:

```
... = ... x ..... // instruction using x
.........         // several instructions not involving x
... = ... x ..... // instruction using x
```

The assembly code would then be

- load x from memory into register; operate on it;
- do the intervening instructions;
- load x from memory into register; operate on it;

With a cache, the assembly code stays the same, but the actual behavior of the memory system now becomes:

- load x from memory into cache, and from cache into register; operate on it;
- do the intervening instructions;
- request x from memory, but since it is still in the cache, load it from the cache into register; operate on it.

Loading from cache is faster than loading from main memory in two ways. The load has a shorter *latency*, meaning the delay between the data being needed in an instruction, and it actually being available. Caches also have a higher *bandwidth*, meaning the number of bytes per second transferred. Thus, if a computation can use data from cache it will be faster than if it relies on data from main memory. Caches are fairly small, so arranging a computation in this manner may take some sophisticated programming.

There is an important difference between cache memory and registers: while data is moved into register by explicit assembly instructions, the move from main memory to cache is entirely done by hardware, invisible to the programmer.

Thus cache use and reuse is outside of direct programmer control. Later, especially in sections 1.6.2 and 6, you will see how it is possible to influence cache use indirectly.

1.3.5.2 Cache hits and misses

Data needed in some operation gets copied into the various caches on its way to the processor. If, some instructions later, a data item is needed again, it is first searched for in the L1 cache; if it is not found

there, it is searched for in the L2 cache; if it is not found there, it is loaded from further cache levels or main memory. Finding data in cache is called a *cache hit*, and not finding it a *cache miss*.

The most common type of cache miss is the 'capacity miss', where more data is accessed than fits the cache. The process of cache data being overwritten by data loaded later is referred to as the earlier data being *flushed*. This is illustrated in figure 1.19.

1.3.5.3 Cache tags

In the above example, the mechanism was left unspecified by which it is found whether an item is present in cache. For this, there is a *tag* for each cache location: sufficient information to reconstruct the memory location that the cache item came from.

1.3.5.4 Cache levels, speed, and size

The caches are called 'level 1' and 'level 2' (or, for short, L1 and L2) cache. Modern processors can have an L3, and sometimes an L4 cache. The L1 and L2 caches are part of the *die*, the processor chip, although for the L2 cache that is a relatively recent development; higher levels cache can be off-chip. Caches can be *private* to a core, as is typically the case for the L1 cache, or *shared* between some or all of the cores.

The L1 cache is small, typically around 16 or 32 Kbyte. Level 2 (and, when present, level 3) cache is more plentiful, up to several megabytes, but it is also slower, with longer latency and lower bandwidth. Figure 1.5 has some representative numbers. Unlike main memory, which is expandable, caches are fixed in size. If a version of a processor exists with a larger cache, it is usually considerably more expensive.

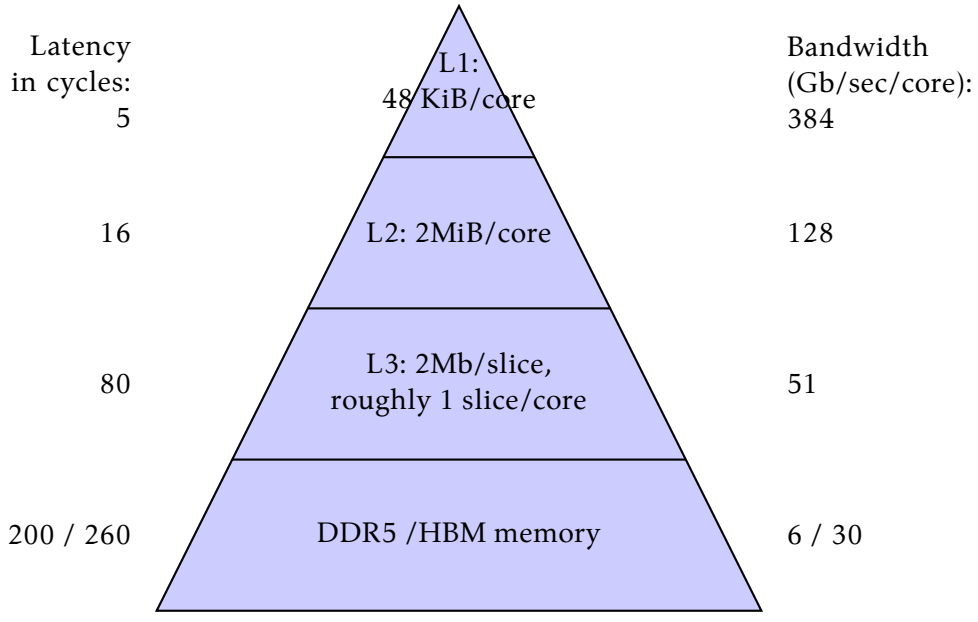


Figure 1.5: Memory hierarchy of an Intel Sapphire Rapids processor

Figure 1.5 illustrates the basic facts of the *cache hierarchy*, in this case for the *Intel Sapphire Rapid* chip: the closer caches are to the FPUs, the faster, but also the smaller they are.

On level 1, there are separate caches for instructions and data; the L2 and L3 cache contain both data and instructions.

You see that the larger caches are increasingly unable to supply data to the processors fast enough. For this reason it is necessary to code in such a way that data is kept as much as possible in the highest cache level possible. We will discuss this issue in detail in the rest of this chapter.

Exercise 1.5. The L1 cache is smaller than the L2 cache, and if there is an L3, the L2 is smaller than the L3. Give a practical and a theoretical reason why this is so.

Experimental exploration of cache sizes and their relation to performance is explored in sections 6.4.1 and 29.2.

1.3.5.5 Types of cache misses

There are three types of cache misses.

As you saw in the example above, the first time you reference data you will always incur a cache miss. This is known as a *compulsory cache miss* since these are unavoidable. Does that mean that you will always be waiting for a data item, the first time you need it? Not necessarily: section 1.3.6 explains how the hardware tries to help you by predicting what data is needed next.

The next type of cache misses is due to the size of your working set: a *capacity cache miss* is caused by data having been overwritten because the cache can simply not contain all your problem data. Figure 1.6

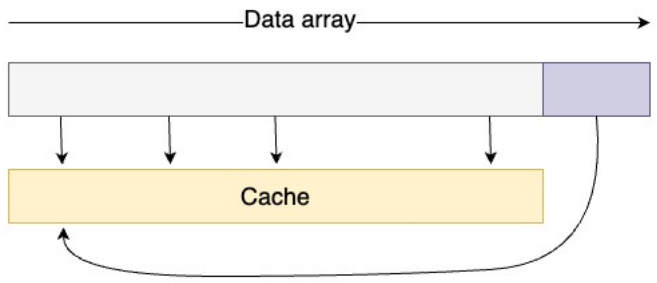


Figure 1.6: Illustration of a capacity miss

illustrates this. (Section 1.3.5.7 discusses how the processor actually decides what data to overwrite.)

If you want to avoid this type of misses, you need to partition your problem in chunks that are small enough that data can stay in cache for an appreciable time. Of course, this presumes that data items are operated on multiple times, so that there is actually a point in keeping it in cache; this is discussed in section 1.6.1.

Finally, there are *conflict misses* caused by one data item being mapped to the same cache location as another, while both are still needed for the computation, and there would have been better candidates to evict. This is discussed in section 1.3.5.11.

In a *multicore* context there is a further type of cache miss: the *invalidation miss*. This happens if an item in cache has become invalid because another core changed the value of the corresponding memory address. The core will then have to reload this address.

1.3.5.6 *Reuse is the name of the game*

The presence of one or more caches is not immediately a guarantee for high performance: this largely depends on the *memory access pattern* of the code, and how well this exploits the caches. The first time that an item is referenced, it is copied from memory into cache, and through to the processor registers. The latency and bandwidth for this are not mitigated in any way by the presence of a cache. When the same item is referenced a second time, it may be found in cache, at a considerably reduced cost in terms of latency and bandwidth: caches have shorter latency and higher bandwidth than main memory.

We conclude that, first, an algorithm has to have an opportunity for data reuse. If every data item is used only once (as in addition of two vectors), there can be no reuse, and the presence of caches is largely irrelevant. A code will only benefit from the increased bandwidth and reduced latency of a cache if items in cache are referenced more than once; see section 1.6.1 for a detailed discussion. An example would be the matrix-vector multiplication $y = Ax$ where each element of x is used in n operations, where n is the matrix dimension; see section 6.10.

Secondly, an algorithm may theoretically have an opportunity for reuse, but it needs to be coded in such a way that the reuse is actually exposed. We will address these points in section 1.6.2. This second point especially is not trivial.

Some problems are small enough that they fit completely in cache, at least in the L3 cache. This is something to watch out for when *benchmarking*, since it gives a too rosy picture of processor performance.

1.3.5.7 *Replacement policies*

Data in cache is placed there by the system, outside of programmer control. Likewise, the system decides when to overwrite data in the cache if it is not referenced in a while, and as other data needs to be placed there. Below, we will go into detail on how caches do this, but as a general principle, a Least Recently Used (LRU) cache replacement policy is used: if a cache is full and new data needs to be placed into it, the data that was least recently used is *flushed* (sometimes: *evicted*) from cache. meaning that it is overwritten with the new data, and is therefore no longer accessible. LRU is by far the most common replacement policy; other possibilities are FIFO (first in first out) or random replacement.

Exercise 1.6. How does the LRU replacement policy related to direct-mapped versus associative caches?

Exercise 1.7. Sketch a simple scenario, and give some (pseudo) code, to argue that LRU is preferable over FIFO as a replacement strategy.

1.3.5.8 *cachelines*

Data movement between memory and cache, or between caches, is not done in single bytes, or even words. Instead, the smallest unit of data moved is called a *cacheline*, sometimes called a *cache block*. A typical cacheline is 64 or 128 bytes long, which in the context of scientific computing implies 8 or 16 double precision floating point numbers. The cacheline size for data moved into L2 cache can be larger than for data moved into L1 cache.

A first motivation for cachelines is a practical one of simplification: if a cacheline is 64 bytes long, six fewer bits need to be specified to the circuitry that moves data from memory to cache. Secondly, cachelines make

sense since many codes show *spatial locality*: if one word from memory is needed, there is a good chance that adjacent words will be pretty soon after.

Conversely, there is now a strong incentive to code in such a way to exploit this locality, since any memory access costs the transfer of several words An efficient program then tries to use the other items on the cacheline, since access to them is effectively free. This phenomenon is visible in code that accesses arrays by *stride*: elements are read or written at regular intervals.

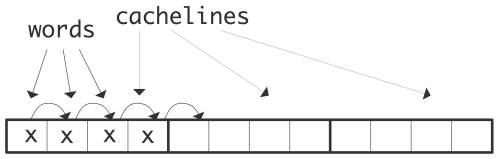


Figure 1.7: Accessing 4 elements at stride 1.

Stride 1 corresponds to sequential access of an array:

```
for (int i=0; i<N; ++i)
  ... = ... x[i] ...
```

In figure 1.7 we have illustrated this case with 4 words per cacheline. Requesting the first elements loads the whole cacheline that contains it into cache. A request for the 2nd, 3rd, and 4th element can then be satisfied from cache, meaning with high bandwidth and low latency.

A larger stride, illustrated in figure 1.8

```
for (int i=0; i<N; i+=stride)
  ... = ... x[i] ...
```

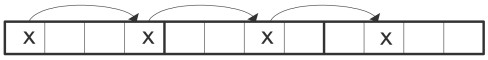


Figure 1.8: Accessing 4 elements at stride 3.

implies that in every cacheline only certain elements are used. We illustrate that with stride 3: requesting the first element loads a cacheline, and this cacheline also contains the second element. However, the third element is on the next cacheline, so loading this incurs the latency and bandwidth of main memory. The same holds for the fourth element. Loading four elements now needs loading three cachelines instead of one, meaning that two-thirds of the available bandwidth has been wasted. (This second case would also incur three times the latency of the first, if it weren't for the hardware *prefetch* mechanism that notices the regular access patterns, and pre-emptively loads further cachelines; see section 1.3.6.)

Some applications naturally lead to strides greater than 1, for instance, accessing only the real parts of an array of complex numbers (for some remarks on the practical realization of complex numbers see section 3.8.6). Also, methods that use *recursive doubling* often have a code structure that exhibits non-unit strides

```
for (int i=0; i<N/2; ++i)
  x[i] = y[2*i];
```

In this discussion of cachelines, we have implicitly assumed the beginning of a cacheline is also the beginning of a word, be that an integer or a floating point number. This need not be true: an 8-byte floating point number can be placed straddling the boundary between two cachelines. You can image that this is not good for performance. Section 29.1.3 discusses ways to address *cacheline boundary alignment* in practice.

1.3.5.9 Cache mapping

Caches get faster, but also smaller, the closer to the FPUs they get, yet even the largest cache is considerably smaller than the main memory size. In section 1.3.5.7 we have already discussed how the decision is made which elements to keep and which to replace.

We will now address the issue of *cache mapping*, which is the question of 'if an item is placed in cache, where does it get placed'. This problem is generally addressed by mapping the (main memory) address of the item to an address in cache, leading to the question 'what if two items get mapped to the same address'.

1.3.5.10 Direct mapped caches

The simplest cache mapping strategy is *direct mapping*. Suppose that memory addresses are 32 bits long, so that they can address 4G bytes[2]; suppose further that the cache has 8K words, that is, 64K bytes, needing 16 bits to address. Direct mapping then takes from each memory address the last ('least significant') 16 bits, and uses these as the address of the data item in cache; see figure 1.9.

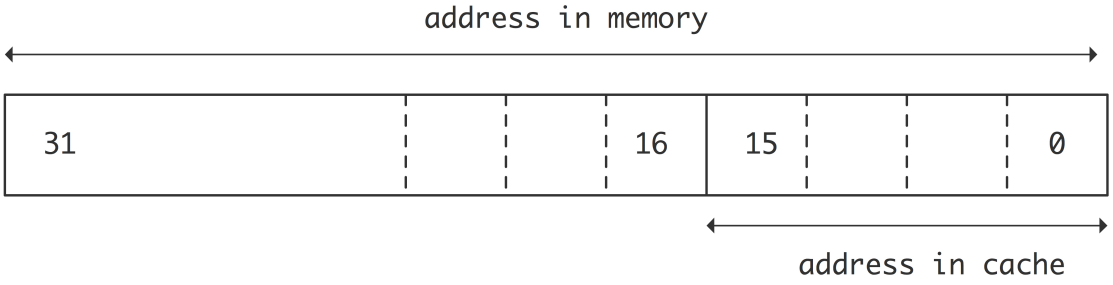


Figure 1.9: Direct mapping of 32-bit addresses into a 64K cache.

Direct mapping is very efficient because its address calculations can be performed very quickly, leading to low latency, but it has a problem in practical applications. If two items are addressed that are separated by 8K words, they will be mapped to the same cache location, which will make certain calculations inefficient. Example:

```
double A[3][8192];
for (int i=0; i<512; ++i)
  a[2][i] = ( a[0][i]+a[1][i] )/2.;
```

or in Fortran:

```
real*8 A(8192,3);
do i=1,512
  a(i,3) = ( a(i,1)+a(i,2) )/2
end do
```

2. We implicitly use the convention that K,M,G suffixes refer to powers of 2 rather than 10: 1K=1024, 1M=1,048,576, 1G=1,073,741,824.

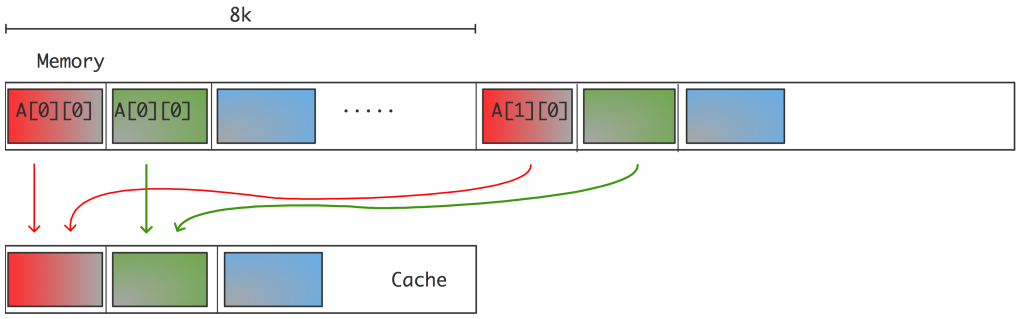


Figure 1.10: Mapping conflicts in direct mapped cache.

Here, the locations of `a[0][i]`, `a[1][i]`, and `a[2][i]` (or `a(i,1)`,`a(i,2)`,`a(i,3)`) are 8K from each other for every i, so the last 16 bits of their addresses will be the same, and hence they will be mapped to the same location in cache; see figure 1.10.

The execution of the loop will now go as follows:

- The data at `a[0][0]` is brought into cache and register. This engenders a certain amount of latency. Together with this element, a whole cacheline is transferred.
- The data at `a[1][0]` is brought into cache (and register, as we will not remark anymore from now on), together with its whole cacheline, at cost of some latency. Since this cache line is mapped to the same location as the first, the first cache line is overwritten.
- In order to write the output, the cacheline containing `a[2][0]` is brought into memory. This is again mapped to the same location, causing flushing of the cacheline just loaded for `a[1][0]`.
- In the next iteration, `a[0][1]` is needed, which is on the same cacheline as `a[0][0]`. However, this cacheline has been flushed, so it needs to be brought in anew from main memory or a deeper cache level. In doing so, it overwrites the cacheline that holds `a[2][0]`.
- A similar story holds for `a[1][1]`: it is on the cache line of `a[1][0]`, which unfortunately has been overwritten in the previous step.

If a cacheline holds four words, we see that each four iterations of the loop involve eight transfers of elements of `a`, where two would have sufficed, if it were not for the cache conflicts.

Exercise 1.8. In the example of direct mapped caches, mapping from memory to cache was done by using the final 16 bits of a 32 bit memory address as cache address. Show that the problems in this example go away if the mapping is done by using the first ('most significant') 16 bits as the cache address. Why is this not a good solution in general?

Remark *So far, we have pretended that caching is based on virtual memory addresses. In reality, caching is based on* physical addresses *of the data in memory, which depend on the algorithm mapping virtual addresses to* memory pages.

1.3.5.11 Associative caches

The problem of cache conflicts would be solved if any data item could go to any cache location. In that case there would be no conflicts, other than the cache filling up, in which case a cache replacement policy (section 1.3.5.7) would flush data to make room for any incoming item. This is called a *fully associative*

cache, and while this design seems optimal, it is also very costly to build, and slower in use than a direct mapped cache.

For this reason, the most common solution is to have a k-way *associative cache*, where k is at least two. In this case, a data item can go to any of k cache locations.

Code would have to have a $k + 1$-way conflict before data would be flushed as in the above example. In that example, a value of $k = 3$ would suffice, but in practice higher values are often encountered. Figure 1.11 illustrates the mapping of memory addresses to cache locations for a direct mapped and a

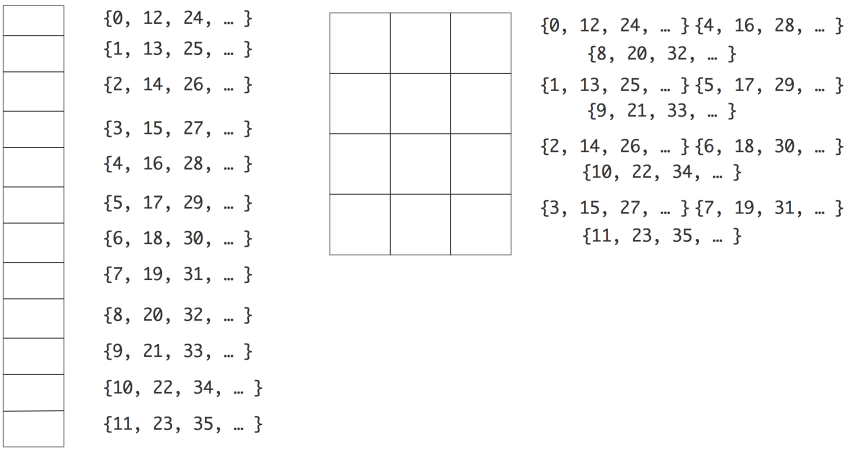


Figure 1.11: Two caches of 12 elements: direct mapped (left) and 3-way associative (right).

3-way associative cache. Both caches have 12 elements, but these are used differently. The direct mapped cache (left) will have a conflict between memory address 0 and 12, but in the 3-way associative cache these two addresses can be mapped to any of three elements.

As a practical example, the *Intel Ice Lake* processor has an L1 cache of 48K bytes that is 12-way set associative with a 64 byte cacheline size, and an L2 cache of 1.25M bytes that is 20-way set associative with a 64 byte cacheline size. The L1 structure is depicted in figure 1.12.

Exercise 1.9. Write a small cache simulator in your favorite language. Assume a k-way associative cache of 32 entries and an architecture with 16 bit addresses. Run the following experiment for $k = 1, 2, 4, \ldots$:

1. Let k be the associativity of the simulated cache.
2. Write the translation from 16 bit memory addresses to $32/k$ cache addresses.
3. Generate 32 random machine addresses, and simulate storing them in cache.

Since the cache has 32 entries, optimally the 32 addresses can all be stored in cache. The chance of this actually happening is small, and often the data of one address will be *evicted* from the cache (meaning that it is overwritten) when another address conflicts with it. Record how many addresses, out of 32, are actually stored in the cache at the end of the simulation. Do step 3 100 times, and plot the results; give median and average value, and the standard deviation. Observe that increasing the

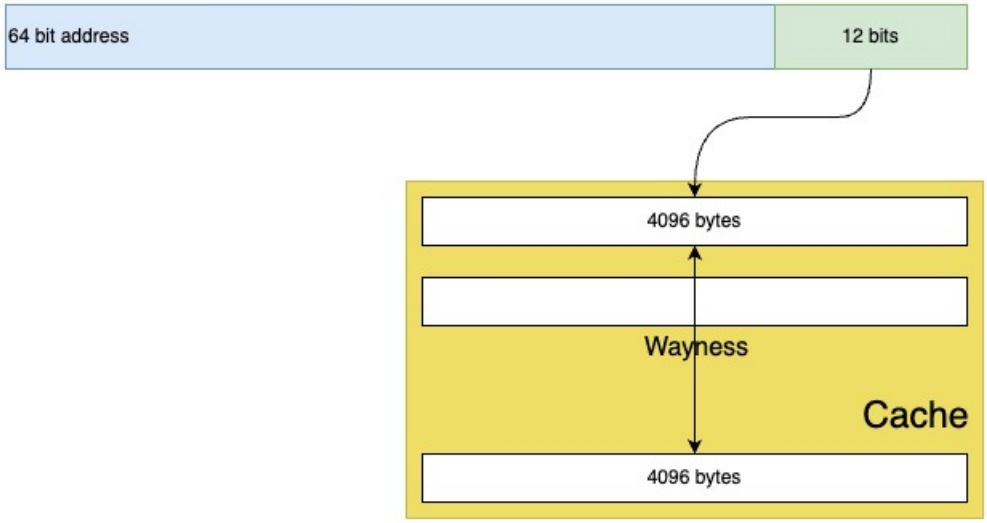


Figure 1.12: Associative cache structure

associativity improves the number of addresses stored. What is the limit behavior? (For bonus points, do a formal statistical analysis.)

1.3.5.12 *Cache memory versus regular memory*

So what's so special about cache memory; why don't we use its technology for all of memory?

Caches typically consist of *Static Random-Access Memory (SRAM)*, which is faster than *Dynamic Random-Access Memory (DRAM)* used for the main memory, but is also more expensive, taking 5–6 transistors per bit rather than one, and it draws more power.

1.3.5.13 *Loads versus stores*

In the above description, all data accessed in the program needs to be moved into the cache before the instructions using it can execute. This holds both for data that is read and data that is written. However, data that is written, and that will not be needed again (within some reasonable amount of time) has no reason for staying in the cache, potentially creating conflicts or evicting data that can still be reused. For this reason, compilers often have support for *streaming stores*: a contiguous stream of data that is purely output will be written straight to memory, without being cached.

1.3.6 Prefetch streams

In the traditional von Neumann model (section 1.1) each instruction contains the location of its operands, so a CPU implementing this model would make a separate request for each new operand. In practice, subsequent data requests are often adjacent or regularly spaced in memory. The memory system can try to detect such data patterns by looking at cache miss points, and request a *prefetch data stream*; figure 1.13.

In its simplest form, the CPU will detect that consecutive loads come from two consecutive cachelines, and automatically issue a request for the next following cacheline. This process can be repeated or extended

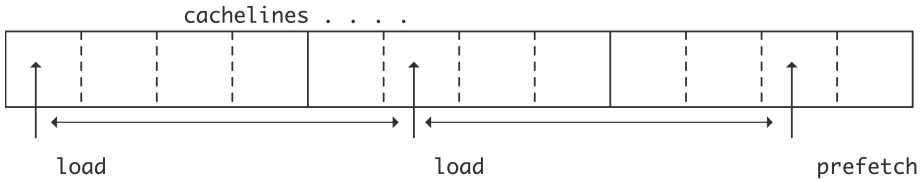


Figure 1.13: Prefetch stream generated by equally spaced requests.

if the code makes an actual request for that third cache line. Since these cachelines are now brought from memory well before they are needed, prefetch has the possibility of eliminating the latency for all but the first couple of data items.

Prefetch streams typically can not cross *small page* boundaries, so it pays to have data be aligned on small page boundaries. Typically this means having a starting address at a multiple of 4K; see section 1.3.9. Some codes benefit from larger page sizes, for instance of a megabyte. This can be set at the Operating System (OS) level.

The concept of *cache miss* now needs to be revisited a little. From a performance point of view we are only interested in *stall*s on cache misses, that is, the case where the computation has to wait for the data to be brought in. Data that is not in cache, but can be brought in while other instructions are still being processed, is not a problem. If an 'L1 miss' is understood to be only a 'stall on miss', then the term 'L1 cache refill' is used to describe all cacheline loads, whether the processor is stalling on them or not.

1.3.6.1 Prefetch instructions

Since prefetch is controlled by the hardware, it is also described as *hardware prefetch*. Prefetch streams can sometimes be controlled from software, for instance through *intrinsics*.

Introducing prefetch by the programmer is a careful balance of a number of factors [92]. Prime among these is the *prefetch distance*: the number of cycles between the start of the prefetch and when the data is needed. In practice, this is often the number of iterations of a loop: the prefetch instruction requests data for a future iteration.

1.3.7 Concurrency and memory transfer

In the discussion about the memory hierarchy we made the point that memory is slower than the processor. As if that is not bad enough, it is not even trivial to exploit all the bandwidth that memory offers. In other words, if you don't program carefully you will get even less performance than you would expect based on the available bandwidth. Let's analyze this.

The memory system typically has a bandwidth of more than one floating point number per cycle, so you need to issue that many requests per cycle to utilize the available bandwidth. This would be true even with zero latency; since there is latency, it takes a while for data to make it from memory and be processed. Consequently, any data requested based on computations on the first data has to be requested with a delay at least equal to the memory latency.

For full utilization of the bandwidth, at all times a volume of data equal to the bandwidth times the latency has to be in flight. Since these data have to be independent, we get a statement of *Little's law* [142]:

$$\text{Concurrency} = \text{Bandwidth} \times \text{Latency}.$$

This is illustrated in figure 1.14. The problem with maintaining this concurrency is not that a program does

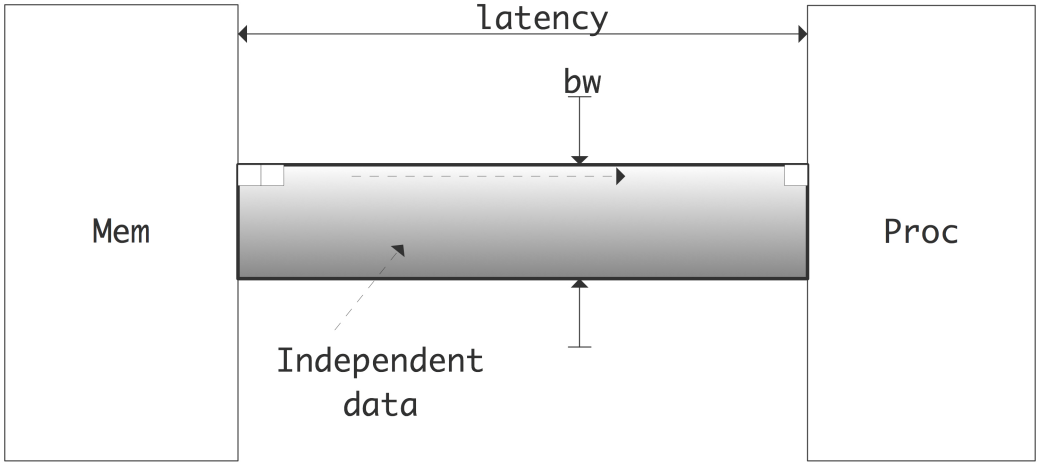


Figure 1.14: Illustration of Little's Law that states how much independent data needs to be in flight.

not have it; rather, the problem is to get the compiler and runtime system to recognize it. For instance, if a loop traverses a long array, the compiler will not issue a large number of memory requests. The prefetch mechanism (section 1.3.6) will issue some memory requests ahead of time, but typically not enough. Thus, in order to use the available bandwidth, multiple streams of data need to be under way simultaneously. Therefore, we can also phrase Little's law as

$$\text{Effective throughput} = \text{Expressed concurrency}/\text{Latency}.$$

1.3.8 Memory banks

Above, we discussed issues relating to bandwidth. You saw that memory, and to a lesser extent caches, have a bandwidth that is less than what a processor can maximally absorb. The situation is actually even worse than the above discussion made it seem. For this reason, memory is often divided into *memory banks* that are interleaved: with four memory banks, words $0, 4, 8, \ldots$ are in bank 0, words $1, 5, 9, \ldots$ are in bank 1, et cetera.

Suppose we now access memory sequentially, then such 4-way interleaved memory can sustain four times the bandwidth of a single memory bank. Unfortunately, accessing by stride 2 will halve the bandwidth, and larger strides are even worse. If two consecutive operations access the same memory bank, we speak of a *bank conflict* [6]. In practice the number of memory banks will be higher, so that strided memory access with small strides will still have the full advertised bandwidth. For instance, the *Cray-1* had 16 banks, and the *Cray-2* 1024.

Exercise 1.10. Show that with a prime number of banks, any stride up to that number will be conflict free. Why do you think this solution is not adopted in actual memory architectures?

In modern processors, *DRAM* still has banks, but the effects of this are felt less because of the presence of caches. However, *GPUs* have memory banks and no caches, so they suffer from some of the same problems as the old supercomputers.

Exercise 1.11. The *recursive doubling* algorithm for summing the elements of an array is:

```
for (int s=2; s<2*n; s*=2)
  for (int i=0; i<n-s/2; i+=s)
    x[i] += x[i+s/2]
```

Analyze bank conflicts for this algorithm. Assume $n = 2^p$ and banks have 2^k elements where $k < p$. Also consider this as a parallel algorithm where all iterations of the inner loop are independent, and therefore can be performed simultaneously.
Alternatively, we can use *recursive halving*:

```
for (int s=(n+1)/2; s>1; s/=2)
  for (int i=0; i<n; i+=1)
    x[i] += x[i+s]
```

Again analyze bank conficts. Is this algorithm better? In the parallel case?

Cache memory can also use banks. For instance, the cachelines in the L1 cache of the *AMD Barcelona* chip are 16 words long, divided into two interleaved banks of 8 words. This means that sequential access to the elements of a cacheline is efficient, but strided access suffers from a deteriorated performance.

1.3.9 TLB, pages, and virtual memory

All of a program's data may not be in memory simultaneously. This can happen for a number of reasons:

- The computer serves multiple users, so the memory is not dedicated to any one user;
- The computer is running multiple programs, which together need more than the physically available memory;
- One single program can use more data than the available memory.

For this reason, computers use *virtual memory*: if more memory is needed than is available, certain blocks of memory are written to disc. In effect, the disc acts as an extension of the real memory. This means that a block of data can be anywhere in memory, and in fact, if it is *swapped* in and out, it can be in different locations at different times. Swapping does not act on individual memory locations, but rather on *memory pages*: contiguous blocks of memory, from a few kilobytes to megabytes in size. (In an earlier generation of operating systems, moving memory to disc was a programmer's responsibility. Pages that would replace each other were called *overlays*.)

For this reason, we need a translation mechanism from the memory addresses that the program uses to the actual addresses in memory. This translation has to be dynamic: if a program is run twice, the same data can be mapped to two different locations. A program has a 'logical data space' (typically starting from address zero) of the addresses used in the compiled code, and this is translated during program execution to actual memory addresses. For this reason, there is a *page table* that specifies which memory pages contain which logical pages.

1.3.9.1 Large pages

In very irregular applications, for instance databases, the page table can get very large as more-or-less random data is brought into memory. However, sometimes these pages show some amount of clustering, meaning that if the page size had been larger, the number of needed pages would be greatly reduced. For this reason, operating systems can have support for *large pages*, typically of size around 2 Mb. (Sometimes 'huge pages' are used; for instance the *Intel Knights Landing* has Gigabyte pages.)

The benefits of large pages are application-dependent: if the small pages have insufficient clustering, use of large pages may fill up memory prematurely with the unused parts of the large pages.

1.3.9.2 TLB

However, address translation by lookup in this table is slow, so CPUs have a *Translation Look-aside Buffer (TLB)*. The TLB is a cache of frequently used page table entries: it provides fast address translation for a number of pages.

If a program needs a memory location, the TLB is consulted to see whether this location is in fact on a page that is remembered in the TLB. If this is the case, the logical address is translated to a physical one; this is a very fast process. The case where the page is not remembered in the TLB is called a *TLB miss*, and the page lookup table is then searched. This is another case of *spatial locality*: accessing (many) elements that are spread far apart incurs a performance penalty from TLB misses.

A typical TLB has between 64 and 512 entries. If a program accesseses data sequentially, it will typically alternate between just a few pages, and there will be no TLB misses. On the other hand, a program that accesses many random memory locations can experience a slowdown because of such misses. The set of pages that is in current use is called the 'working set'.

Section 6.4.4 and appendix 29.5 discuss some simple code illustrating the behavior of the TLB.

[There are some complications to this story. For instance, there is usually more than one TLB. The first one is associated with the L2 cache, the second one with the L1. In the *AMD Opteron*, the L1 TLB has 48 entries, and is fully (48-way) associative, while the L2 TLB has 512 entries, but is only 4-way associative. This means that there can actually be TLB conflicts. In the discussion above, we have only talked about the L2 TLB. The reason that this can be associated with the L2 cache, rather than with main memory, is that the translation from memory to L2 cache is deterministic.]

Use of *large pages* also reduces the number of potential TLB misses, since the working set of pages can be reduced.

1.4 Multicore architectures

In recent years, the limits of performance have been reached for the traditional processor chip design.

- Clock frequency can not be increased further, since it increases energy consumption, heating the chips too much; see section 1.7.1.

- It is not possible to extract more Instruction Level Parallelism (ILP) from codes, either because of compiler limitations, because of the limited amount of intrinsically available parallelism, or because branch prediction makes it impossible (see section 1.2.5).

One of the ways of getting a higher utilization out of a single processor chip is then to move from a strategy of further sophistication of the single processor to a division of the chip into multiple processing *cores*. The separate cores can work on unrelated tasks, or, by splitting a large amount of data parallelism into parallel tasks collaborate on a common task at a higher overall performance [155]. **Remark** *Another solution is Intel's* hyperthreading, *which lets a processor mix the instructions of several instruction streams. The benefits of this are strongly dependent on the individual case. However, this same mechanism is exploited with great success in GPUs; see section 2.9.3. For a discussion see section 2.6.1.9.*

This solves the above two problems:

- Two cores at a lower frequency can have the same throughput as a single processor at a higher frequency; hence, multiple cores are more energy-efficient.
- Discovered ILP is now replaced by explicit task parallelism, managed by the programmer.

While the first multicore CPUs were in effect two processors on the same die, later generations incorporated L3 or L2 caches that were shared between the two processor cores; see figure 1.15. This design

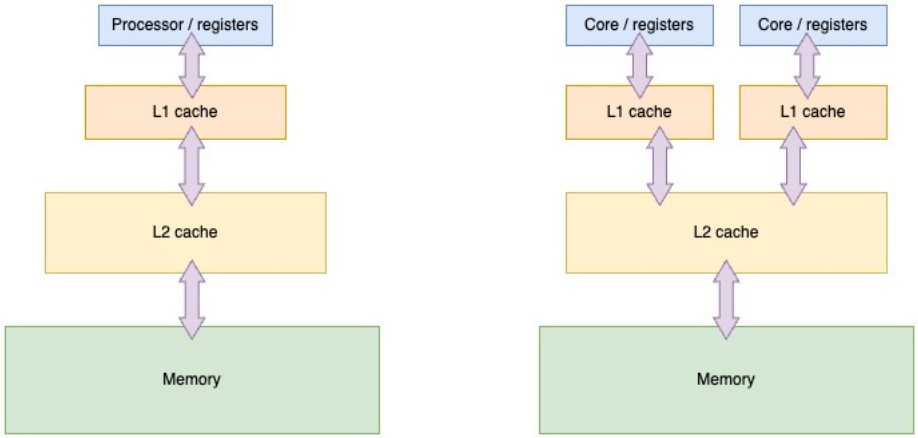


Figure 1.15: Cache hierarchy in a single-core and dual-core chip.

makes it efficient for the cores to work jointly on the same problem. The cores would still have their own L1 cache, and these separate caches lead to a *cache coherence* problem; see section 1.4.1 below.

We note that the term 'processor' is now ambiguous: it can refer to either the chip, or the processor core on the chip. For this reason, we mostly talk about a *socket* for the whole chip and *core* for the part containing one arithmetic and logic unit and having its own registers. Currently, CPUs with dozens of cores are common in supercomputers. Such chips can have complicated cache structures where an L2 cache is neither private, nor globally shared, but instead shared by an 8-core 'chiplet'; see figure 1.16 for an example.

With this mix of shared and private caches, the programming model for multicore processors is becoming a hybrid between shared and distributed memory:

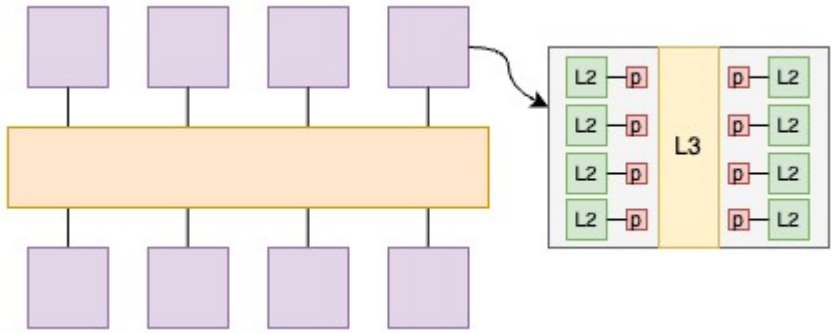


Figure 1.16: Structure of the AMD multicore processors

Core The cores have their own private L1 cache, which is a sort of distributed memory.
Socket On one socket, there is often a shared L2 cache, which is shared memory between all or some of the cores.
Node There can be multiple sockets on a single 'node' or motherboard, accessing the same shared memory.
Network Distributed memory programming (see the next chapter) is needed to let nodes communicate.

Historically, multicore architectures have a precedent in multiprocessor shared memory designs (section 2.4.1) such as the *Sequent Symmetry* and the *Alliant FX/8*. Conceptually the program model is the same, but the technology now allows to shrink a multiprocessor board to a multicore chip.

1.4.1 Cache coherence

With parallel processing, there is the potential for a conflict if more than one processing element has a copy of the same data item. The problem of ensuring that all cached data are an accurate copy of main memory is referred to as *cache coherence*: if the copy in one cache is altered, any other copy needs to be updated too, or at least marked as invalid.

In distributed memory architectures, a dataset is usually partitioned disjointly over the processors, so conflicting copies of data can only arise with knowledge of the user, and it is up to the user to deal with the problem. The case of shared memory is more subtle: since processes access the same main memory, it would seem that conflicts are in fact impossible. However, cores typically have some private cache that contains copies of data from memory, so conflicting copies can occur.

Suppose that two cores have a copy of the same data item in their (private) L1 cache, and one modifies its copy. Now the other has cached data that is no longer an accurate copy of its counterpart: the processor will *invalidate* that copy of the item, and in fact its whole cacheline. When the process needs access to the item again, it needs to reload that cacheline. The alternative is for any core that alters data to send that cacheline to the other cores. This strategy probably has a higher overhead, since other cores are not likely to have a copy of a cacheline.

This process of updating or invalidating cachelines is known as *maintaining cache coherence*, and it is done on a very low level of the processor, with no programmer involvement. However, it will slow down the computation, and it uses bandwidth that could otherwise be used for loading or storing operands.

The state of a cacheline with respect to a data item in main memory is usually described in terms of a protocol such as the Modified-Shared-Invalid (MSI) coherence protocol, where a cacheline can be in the following states on a given core:

Modified: the cacheline has been modified, and needs to be written to the backing store. This writing can be done when the line is evicted, or it is done immediately, depending on the write-back policy.

Shared: the line is present in at least one cache and is unmodified.

Invalid: the line is not present in the current cache, or it is present but a copy in another cache has been modified.

These states control the movement of cachelines between memory and the caches. For instance, suppose a core does a read to a cacheline that is invalid on that core. It can then load it from memory or get it from another cache, which may be faster. (Finding whether a line exists in state M or S on another cache, is called *snooping*; an alternative is to maintain cache directories; see below.) If the line is Shared, it can now simply be copied; if it is in state M in the other cache, that core first needs to write it back to memory.

Exercise 1.12. Consider two processors, a data item x in memory, and cachelines x_1,x_2 in the private caches of the two processors to which x is mapped. Describe the transitions between the states of x_1 and x_2 under reads and writes of x on the two processors. Also indicate which actions cause memory bandwidth to be used.

Variants of the MSI protocol add an 'Exclusive' or 'Owned' state for increased efficiency.

1.4.1.1 Solutions to cache coherence

There are two basic mechanisms for realizing cache coherence: snooping and directory-based schemes.

In the *snooping* mechanism, any request for data is sent to all caches, and the data is returned if it is present anywhere; otherwise it is retrieved from memory. In a variation on this scheme, a core 'listens in' on all bus traffic, so that it can invalidate or update its own cacheline copies when another core modifies its copy. Invalidating is cheaper than updating since it is a bit operation, while updating involves copying the whole cacheline.

Exercise 1.13. When would updating pay off? Write a simple cache simulator to evaluate this question.

Since snooping often involves broadcast information to all cores, it does not scale beyond a small number of cores. A solution that scales better is using a *tag directory*: a central directory that contains the information on what data is present in some cache, and what cache it is in specifically. For processors with large numbers of cores (such as the *Intel Xeon Phi*) the directory can be distributed over the cores.

1.4.1.2 Tag directories

In multicore processors with distributed, but coherent, caches (such as the *Intel Xeon Phi*) the *tag directories* can themselves be distributed. This increases the latency of cache lookup.

1.4.2 False sharing

The cache coherence problem can even appear if the cores access different items. For instance, a declaration

```
  double x,y;
```

will likely allocate x and y next to each other in memory, so there is a high chance they fall on the same cacheline. Now if one core updates x and the other y, this cacheline will continuously be moved between the cores. This is called *false sharing*.

The most common case of false sharing happens when threads update consecutive locations of an array. For instance, in the following OpenMP fragment all threads update their own location in an array of partial results:

```
  local_results = new double[num_threads];
#pragma omp parallel
{
  int thread_num = omp_get_thread_num();
  for (int i=my_lo; i<my_hi; ++i)
    local_results[thread_num] = ... f(i) ...
}
// combine local results into global:
global_result = g(local_results)
```

While there is no actual *race condition* (as there would be if the threads all updated the `global_result` variable), this code may have low performance, since the cacheline(s) with the `local_result` array will continuously be invalidated.

That said, false sharing is less of a problem in modern CPUs than it used to be. Let's consider a code that explores the above scheme:

```
// hardware/falsesharing-omp.cxx
  for (int spacing : {16,12,8,4,3,2,1,0} ) {
    /* ... */
      int iproc = omp_get_thread_num();
    /* ... */
      floattype *write_addr = results.data() + iproc*spacing;
    /* ... */
      for (int r=0; r<how_many_repeats; r++) {
        for (int w=0; w<stream_length; w++) {
          *write_addr += *( read_stream+w );
        }
      }
```

Executing this on the *Intel Core i5* of an *Apple Macbook Air* shows a modest performance degradation:

```
executing: OMP_NUM_THREADS=4 OMP_PROC_BIND=true ./falsesharing-omp
Running with 4 threads; each doing stream length=9000
Spacing 16.. ran for  1393069 usec, equivalent serial time: 1.39307e+06 usec (efficiency: 100% ); operation
     count: 3.6e+09
Spacing 12.. ran for  1337529 usec, equivalent serial time: 1.33753e+06 usec (efficiency: 104% ); operation
     count: 3.6e+09
Spacing  8.. ran for  1307244 usec, equivalent serial time: 1.30724e+06 usec (efficiency: 106% ); operation
     count: 3.6e+09
Spacing  4.. ran for  1368394 usec, equivalent serial time: 1.36839e+06 usec (efficiency: 101% ); operation
     count: 3.6e+09
Spacing  3.. ran for  1603778 usec, equivalent serial time: 1.60378e+06 usec (efficiency:  86% ); operation
     count: 3.6e+09
Spacing  2.. ran for  2044134 usec, equivalent serial time: 2.04413e+06 usec (efficiency:  68% ); operation
     count: 3.6e+09
```

```
Spacing  1.. ran for  1819370 usec, equivalent serial time: 1.81937e+06 usec (efficiency:  76% ); operation
     count: 3.6e+09
Spacing  0.. ran for  1811778 usec, equivalent serial time: 1.81178e+06 usec (efficiency:  76% ); operation
     count: 3.6e+09
```

On the other hand, on the *Intel Cascade Lake* of the *TACC Frontera cluster* we see no such thing:

```
executing: OMP_NUM_THREADS=24 OMP_PROC_BIND=close OMP_PLACES=cores  ./falsesharing-omp
Running with 24 threads; each doing stream length=9000
Spacing 16.. ran for 0.001127 usec, equivalent serial time:       1127 usec (efficiency:   100% ); operation
     count: 1.08e+07
Spacing 12.. ran for 0.001103 usec, equivalent serial time:       1103 usec (efficiency: 102.1% ); operation
     count: 1.08e+07
Spacing  8.. ran for 0.001102 usec, equivalent serial time:       1102 usec (efficiency: 102.2% ); operation
     count: 1.08e+07
Spacing  4.. ran for 0.001102 usec, equivalent serial time:       1102 usec (efficiency: 102.2% ); operation
     count: 1.08e+07
Spacing  3.. ran for 0.001105 usec, equivalent serial time:       1105 usec (efficiency: 101.9% ); operation
     count: 1.08e+07
Spacing  2.. ran for 0.001104 usec, equivalent serial time:       1104 usec (efficiency:   102% ); operation
     count: 1.08e+07
Spacing  1.. ran for 0.001103 usec, equivalent serial time:       1103 usec (efficiency: 102.1% ); operation
     count: 1.08e+07
Spacing  0.. ran for 0.001104 usec, equivalent serial time:       1104 usec (efficiency:   102% ); operation
     count: 1.08e+07
```

The reason is that here the hardware caches the accumulator variable, and does not write to memory until the end of the loop. This obviates all problems with false sharing.

We can force false sharing problems by forcing the writeback to memory, for instance with an *OpenMP atomic* directive:

```
executing: OMP_NUM_THREADS=24 OMP_PROC_BIND=close OMP_PLACES=cores  ./falsesharing-omp
Running with 24 threads; each doing stream length=9000
Spacing 16.. ran for  0.01019 usec, equivalent serial time:  1.019e+04 usec (efficiency:   100% ); operation
     count: 1.08e+07
Spacing 12.. ran for  0.01016 usec, equivalent serial time:  1.016e+04 usec (efficiency: 100.2% ); operation
     count: 1.08e+07
Spacing  8.. ran for  0.01016 usec, equivalent serial time:  1.016e+04 usec (efficiency: 100.2% ); operation
     count: 1.08e+07
Spacing  4.. ran for   0.1414 usec, equivalent serial time:  1.414e+05 usec (efficiency:   7.2% ); operation
     count: 1.08e+07
Spacing  3.. ran for   0.1391 usec, equivalent serial time:  1.391e+05 usec (efficiency:   7.3% ); operation
     count: 1.08e+07
Spacing  2.. ran for    0.184 usec, equivalent serial time:   1.84e+05 usec (efficiency:   5.5% ); operation
     count: 1.08e+07
Spacing  1.. ran for   0.2855 usec, equivalent serial time:  2.855e+05 usec (efficiency:   3.5% ); operation
     count: 1.08e+07
Spacing  0.. ran for   0.3542 usec, equivalent serial time:  3.542e+05 usec (efficiency:   2.8% ); operation
     count: 1.08e+07
```

(Of course, this has a considerable performance penalty by itself.)

1.4.3 Computations on multicore chips

There are various ways that a multicore processor can lead to increased performance. First of all, in a desktop situation, multiple cores can actually run multiple programs. More importantly, we can use the parallelism to speed up the execution of a single code. This can be done in two different ways.

The MPI library (section 2.6.3.3) is typically used to communicate between processors that are connected through a network. However, it can also be used in a single multicore processor: the MPI calls then are realized through shared memory copies.

Alternatively, we can use the shared memory and shared caches and program using threaded systems such as OpenMP (section 2.6.2). The advantage of this mode is that parallelism can be much more dynamic, since the runtime system can set and change the correspondence between threads and cores during the program run.

We will discuss in some detail the scheduling of linear algebra operations on multicore chips; section 7.12.

1.4.4 TLB shootdown

Section 1.3.9.2 explained how the TLB is used to cache the translation from logical address, and therefore logical page, to physical page. The TLB is part of the memory unit of the *socket*, so in a multi-socket design, it is possible for a process on one socket to change the page mapping, which makes the mapping on the other incorrect.

One solution to this problem is called *TLB shoot-down*: the process changing the mapping generates an *Inter-Processor Interrupt*, which causes the other processors to rebuild their TLB.

1.5 Node architecture and sockets

In the previous sections we have made our way down through the memory hierarchy, visiting registers and various cache levels, and the extent to which they can be private or shared. At the bottom level of the memory hierarchy is the memory that all cores share. This can range from a few Gigabyte on a lowly laptop to a few Terabyte in some supercomputer centers.

While this memory is shared between all cores, there is some structure to it. This derives from the fact that a cluster *node* can have more than one *socket*, that is, processor chip. The shared memory on the

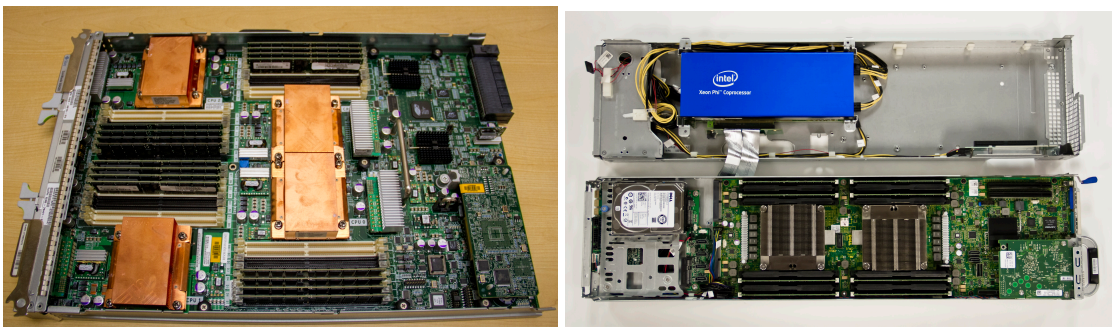


Figure 1.17: Left: a four-socket design. Right: a two-socket design with co-processor.

node is typically spread over banks that are directly attached to one particular socket. This is for instance illustrated in figure 1.17, which shows the four-socket node of the *TACC Ranger cluster* supercomputer

(no longer in production) and the two-socket node of the *TACC Stampede cluster* supercomputer which contains an *Intel Xeon Phi* co-processor. In both designs you clearly see the memory chips that are directly connected to the sockets.

1.5.1 Design considerations

Consider a supercomputer *cluster* to be built-up out of N nodes, each of S sockets, each with C cores. We can now wonder, 'if $S > 1$, why not lower S, and increase N or C?' Such questions have answers as much motivated by price as by performance.

- Increasing the number of cores per socket may run into limitations of *chip fabrication*.
- Increasing the number of cores per socket may also lower the available bandwidth per core. This is a consideration for High-Performance Computing (HPC) applications where cores are likely to be engaged in the same activity; for more heterogeneous workloads this may matter less.
- For fixed C, lowering S and increasing N mostly raises the price because the node price is a weak function of S.
- On the other hand, raising S makes the node architecture more complicated, and may lower performance because of increased complexity of maintaining *coherence* (section 1.4.1). Again, this is less of an issue with heterogeneous workloads, so high values of S are more common in web servers than in HPC installations.

1.5.2 NUMA phenomena

The nodes illustrated above are examples of *Non-Uniform Memory Access (NUMA)* design: for a process running on some core, the memory attached to its socket is slightly faster to access than the memory attached to another socket.

One result of this is the *first-touch* phenomenon. Dynamically allocated memory is not actually allocated until it's first written to. Consider now the following OpenMP (section 2.6.2) code:

```
double *array = (double*)malloc(N*sizeof(double));
for (int i=0; i<N; ++i)
   array[i] = 1;
#pragma omp parallel for
for (int i=0; i<N; ++i)
   .... lots of work on array[i] ...
```

Because of first-touch, the array is allocated completely in the memory of the socket of the main thread. In the subsequent parallel loop the cores of the other socket will then have slower access to the memory they operate on.

The solution here is to also make the initialization loop parallel, even if the amount of work in it may be negligible.

1.6 Locality and data reuse

By now it should be clear that there is more to the execution of an algorithm than counting the operations: the data transfer involved is important, and can in fact dominate the cost. Since we have caches and registers, the amount of data transfer can be minimized by programming in such a way that data stays as close to the processor as possible. Partly this is a matter of programming cleverly, but we can also look at the theoretical question: does the algorithm allow for it to begin with.

It turns out that in scientific computing data often interacts mostly with data that is close by in some sense, which will lead to data locality; section 1.6.2. Often such locality derives from the nature of the application, as in the case of the PDEs you will see in chapter 4. In other cases such as molecular dynamics (chapter 8) there is no such intrinsic locality because all particles interact with all others, and considerable programming cleverness is needed to get high performance.

1.6.1 Data reuse and arithmetic intensity

In the previous sections you learned that processor design is unbalanced in the sense that loading data is slower than executing the actual operations. This imbalance is large for main memory and less for the various cache levels. Thus we are motivated to keep data in cache and keep the amount of *data reuse* as high as possible.

Of course, we need to determine first if the computation allows for data to be reused. For this we define the *arithmetic intensity* of an algorithm as follows:

> If n is the number of data items that an algorithm operates on, and $f(n)$ the number of operations it takes, then the arithmetic intensity is $f(n)/n$.
> Roughly equivalently, we define the *reuse factor* as the number of times each loaded data item is used.

(We can measure data items in either floating point numbers or bytes. The latter possibility makes it easier to relate arithmetic intensity to hardware specifications of a processor.)

Arithmetic intensity is also related to *latency hiding*: the concept that you can mitigate the negative performance impact of data loading behind computational activity going on. For this to work, you need more computations than data loads to make this hiding effective. And that is the very definition of computational intensity: a high ratio of operations per byte/word/number loaded.

1.6.1.1 Example: vector operations

Consider for example the vector addition

$$\forall_i : x_i \leftarrow x_i + y_i.$$

This involves three memory accesses (two loads and one store) and one operation per iteration, giving an arithmetic intensity of $1/3$.

The *Basic Linear Algebra Subprograms (BLAS) AXPY* (for 'a times x plus y') operation

$$\forall_i : x_i \leftarrow a\, x_i + y_i$$

has two operations, but the same number of memory access since the one-time load of a is amortized. It is therefore more efficient than the simple addition, with an arithmetic intensity of $2/3$.

The inner product calculation

$$\forall_i : s \leftarrow s + x_i \cdot y_i$$

is similar in structure to the axpy operation, involving one multiplication and addition per iteration, on two vectors and one scalar. However, now there are only two load operations, since s can be kept in register and only written back to memory at the end of the loop. The arithmetic intensity here is 1.

1.6.1.2 Example: matrix operations

Next, consider the *matrix-matrix product*:

$$\forall_{i,j} : c_{ij} = \sum_k a_{ik} b_{kj}.$$

which is important, for instance in Deep Learning (DL) applications. This operation involves $3n^2$ data items and $2n^3$ operations, which is of a higher order. The arithmetic intensity is $O(n)$, meaning that every data item will be used $O(n)$ times. This has the implication that, with suitable programming, this operation has the potential of overcoming the bandwidth/clock speed gap by keeping data in fast cache memory.

Exercise 1.14. The matrix-matrix product, considered as mathematical operation, clearly has data reuse by the above definition. Argue that this reuse is not trivially attained by a straightforward implementation as a triple loop. What determines whether this implementation has reuse of data that is in cache?

In this discussion we were only concerned with the number of operations of a given implementation, not the mathematical operation. For instance, there are ways of performing the matrix-matrix multiplication and Gaussian elimination algorithms in fewer than $O(n^3)$ operations [182, 160]. However, this requires a different implementation, which has its own analysis in terms of memory access and reuse.

High performance implementation of the matrix-matrix product is discussed in section 7.4.1.

1.6.1.3 Fourier transform

Another algorithm that has potential for data reuse is the Fast Fourier Transform (FFT). Here, $O(N \log N)$ operations are done on $O(N)$ data, so there is an arithmetic intensity of $\log N$. We will not further go into this.

1.6.1.4 The roofline model

There is an elegant way of talking about how arithmetic intensity, which is a statement about how the ideal algorithm, not its implementation, interacts with hardware parameters and the actual implementation to determine performance. This is known as the *roofline model* [197], and it expresses the basic fact that performance is bounded by two factors, illustrated in the first graph of figure 1.18.

1. The *peak performance*, indicated by the horizontal line at the top of the graph, is an absolute bound on the performance, achieved only if every aspect of a CPU (pipelines, multiple floating point units) are perfectly used. The calculation of this number is purely based on CPU properties and clock cycle; it is assumed that memory bandwidth is not a limiting factor.

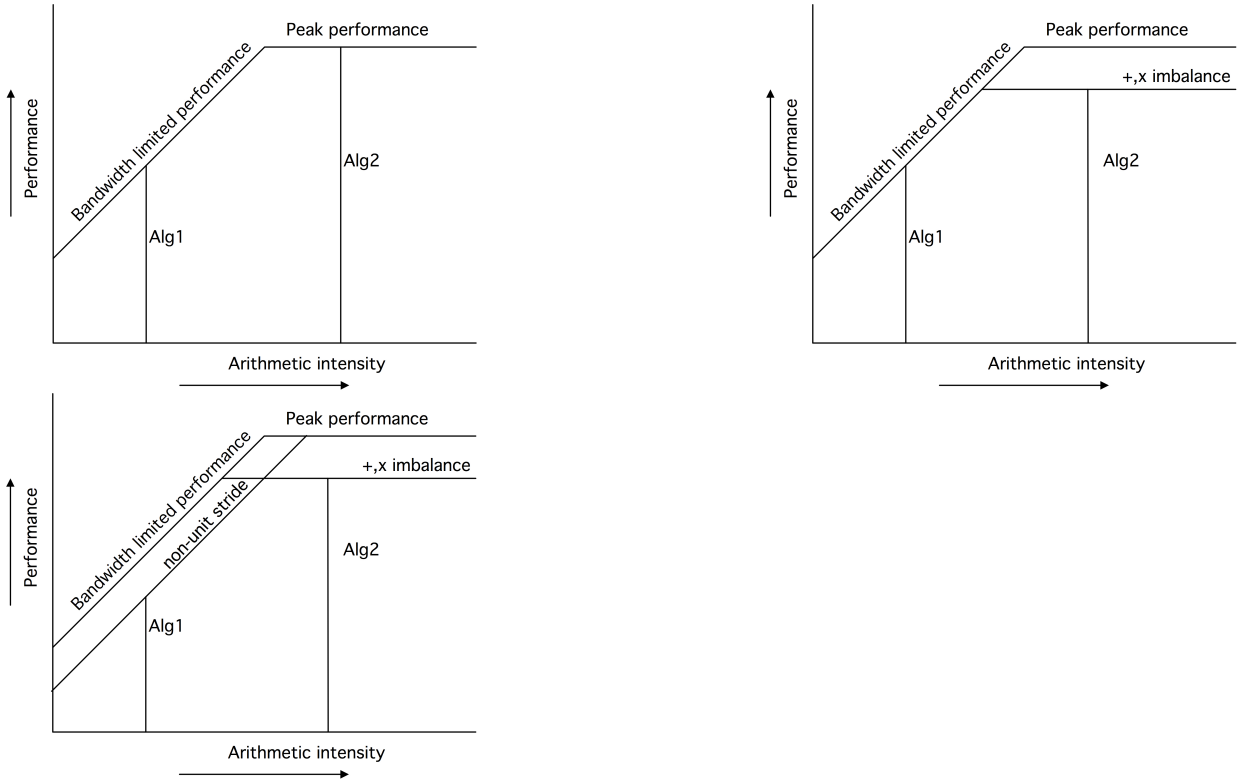


Figure 1.18: Illustration of factors determining performance in the roofline model.

2. The number of operations per second is also limited by the product of the bandwidth, an absolute number, and the arithmetic intensity:

$$\frac{\text{operations}}{\text{second}} = \frac{\text{operations}}{\text{dataitem}} \cdot \frac{\text{dataitems}}{\text{second}}$$

   If the first factor is 1 we have a streaming operation, and performance is limited by bandwidth. This is depicted by the linearly increasing line in the graph.

The roofline model is an elegant way of expressing that various factors lower the ceiling. For instance, if an algorithm fails to use the full *SIMD width*, this imbalance lowers the attainable peak. The second graph in figure 1.18 indicates various factors that lower the ceiling, such as non-unit strides. There are also various factors that lower the available bandwidth, such as imperfect latency hiding. This is indicated by a lowering of the sloping roofline in the third graph.

For a given arithmetic intensity, the performance is determined by where its vertical line intersects the roof line. If this is at the horizontal part, the computation is called *compute-bound*: performance is determined by characteristics of the processor, and bandwidth is not an issue. On the other hand, if that vertical line intersects the sloping part of the roof, the computation is called *bandwidth-bound*: performance is determined by the memory subsystem, and the full capacity of the processor is not used.

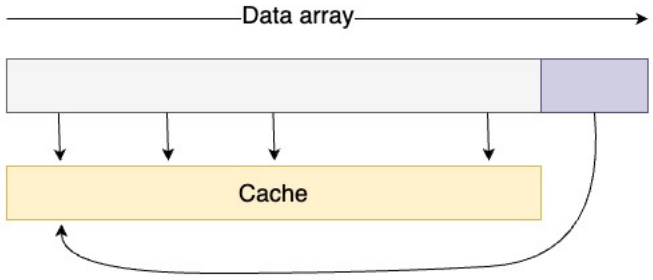


Figure 1.19: Large vector flushes old data from cache

Exercise 1.15. How would you determine whether a given program kernel is bandwidth or compute bound?

1.6.2 Locality

Since using data in cache is faster than getting data from main memory, a programmer obviously wants to maximize cache reuse in order to get higher performance. While placing data in cache is not under explicit programmer control, even from assembly language, in most CPUs it is still possible to know what data is in cache, and to some extent to control it.

In the context of performance optimization, two crucial concepts are *temporal locality* and *spatial locality*. Temporal locality is the easiest to explain: this describes the use of a data element within a short time of its last use. If the second reference comes quickly enough after the first, data will still be in cache and is therefore quicker accessible.

Spatial locality is harder to explain, and we will do so in detail later. The basic idea is that after one data item has been referenced, others that are in some sense 'close to it' will be quicker to load.

1.6.2.1 Temporal locality

As an example of *temporal locality*, consider the repeated traversal of a long vector:

```
for (int loop=0; loop<10; ++loop) {
  for (int i=0; i<N; ++i) {
    ... = ... x[i] ...
  }
}
```

Each element of `x` will be used 10 times, but if the vector (plus other data accessed) exceeds the cache size, each element will be flushed before its next use; see figure 1.19. Therefore, the use of `x[i]` does not exhibit temporal locality: subsequent uses are spaced too far apart in time for an array element to remain in cache.

If the structure of the computation allows us to exchange the loops:

```
for (int i=0; i<N; ++i) {
  for (int loop=0; loop<10; ++loop) {
```

```
    ... = ... x[i] ...
  }
}
```

the elements of x are now repeatedly reused, and are therefore more likely to remain in the cache. This rearranged code displays better temporal locality in its use of `x[i]`. (On the other hand, the original code is more likely to use vector instructions. The loss of these counteracts the performance gain from the cache to an extent.)

1.6.2.2 Spatial locality

The concept of *spatial locality* is slightly more involved. A program is said to exhibit spatial locality if it references memory that is 'close' to memory it already referenced. In the classical von Neumann architecture with only a processor and memory, spatial locality should be irrelevant, since one address in memory can be as quickly retrieved as any other. However, in a modern CPU with caches, the story is different.

Since data is moved in *cache line*s rather than individual words or bytes, there is a great benefit to coding in such a manner that all elements of the cacheline are used. In the loop

```
for (int i=0; i<N*s; i+=s) {
    ... x[i] ...
}
```

spatial locality is a decreasing function of the *stride* s.

Let S be the cacheline size, then as s ranges from $1 \ldots S$, the number of elements used of each cacheline goes down from S to 1. Relatively speaking, this increases the cost of memory traffic in the loop: if $s = 1$, we load $1/S$ cachelines per element; if $s = S$, we load one cacheline for each element. This effect is demonstrated in section 6.2.2.

A second example of spatial locality worth observing involves the TLB (section 1.3.9.2). If a program references elements that are close together, they are likely on the same memory page, and address translation through the TLB will be fast. On the other hand, if a program references many widely disparate elements, it will also be referencing many different pages. The resulting TLB misses are very costly; see also section 6.4.4. Fortunately, TLBs these days are very large so you are unlikely to run into this.

Exercise 1.16. Consider the following pseudocode of an algorithm for summing n numbers $x[i]$ where n is a power of 2:

```
for s=2,4,8,...,n/2,n:
  for i=0 to n-1 with steps s:
    x[i] = x[i] + x[i+s/2]
sum = x[0]
```

Analyze the spatial and temporal locality of this algorithm, and contrast it with the standard algorithm

```
sum = 0
for i=0,1,2,...,n-1
  sum = sum + x[i]
```

Exercise 1.17. Consider the following code, and assume that `nvectors` is small compared to the cache size, and `length` large.

```
for (int k=0; k<nvectors; ++k)
  for (int i=0; i<length; ++i)
    a[k,i] = b[i] * c[k]
```

How do the following concepts relate to the performance of this code:

- Reuse
- Cache size
- Associativity

Would the following code where the loops are exchanged perform better or worse, and why?

```
for (int i=0; i<length; ++i)
  for (int k=0; k<nvectors; ++k)
    a[k,i] = b[i] * c[k]
```

1.6.2.3 *Examples of locality*

Let us examine locality issues for a realistic example. The matrix-matrix multiplication $C \leftarrow A \cdot B$ can be computed in several ways. We compare two implementations, assuming that all matrices are stored by rows, and that the cache size is insufficient to store a whole row or column.

```
for i=1..n
  for j=1..n
    for k=1..n
      c[i,j] += a[i,k]*b[k,j]
```

```
for i=1..n
  for k=1..n
    for j=1..n
      c[i,j] += a[i,k]*b[k,j]
```

These implementations are illustrated in figure 1.20 The first implementation constructs the (i, j) element

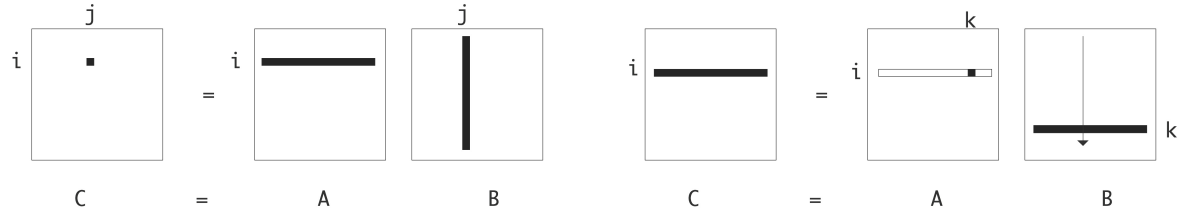


Figure 1.20: Two loop orderings for the $C \leftarrow A \cdot B$ matrix-matrix product.

of C by the inner product of a row of A and a column of B, in the second a row of C is updated by scaling rows of B by elements of A.

Our first observation is that both implementations indeed compute $C \leftarrow C + A \cdot B$, and that they both take roughly $2n^3$ operations. However, their memory behavior, including spatial and temporal locality, is very different.

In the first implementation, `c[i,j]` is invariant in the inner iteration, which constitutes temporal locality, so it can be kept in register. As a result, each element of C will be loaded and stored only once.

In the second implementation, `c[i,j]` will be loaded and stored in each inner iteration. In particular, this implies that there are now n^3 store operations, a factor of n more than in the first implementation.
In both implementations, `a[i,k]` elements are accessed by rows, so there is good spatial locality, as each loaded cacheline will be used entirely. In the second implementation, `a[i,k]` is invariant in the inner loop, which constitutes temporal locality; it can be kept in register. As a result, in the second case A will be loaded only once, as opposed to n times in the first case.
The two implementations differ greatly in how they access the matrix B. First of all, `b[k,j]` is never invariant so it will not be kept in register, and B engenders n^3 memory loads in both cases. However, the access patterns differ.
In second case, `b[k,j]` is accessed by rows, so there is good spatial locality: cachelines will be fully utilized after they are loaded.
In the first implementation, `b[k,j]` is accessed by columns. Because of the row storage of the matrices, a cacheline contains a part of a row, so for each cacheline loaded, only one element is used in the columnwise traversal. This means that the first implementation has more loads for B by a factor of the cacheline length. There may also be TLB effects.

Note that we are not making any absolute predictions on code performance for these implementations, or even relative comparison of their runtimes. Such predictions are very hard to make. However, the above discussion identifies issues that are relevant for a wide range of classical CPUs.

`c[i,j] a[i,k]` **Exercise** 1.18. There are more algorithms for computing the product $C \leftarrow A \cdot B$. Consider the following:

```
for k=1..n:
  for i=1..n:
    for j=1..n:
      c[i,j] += a[i,k]*b[k,j]
```

Analyze the memory traffic for the matrix C, and show that it is worse than the two algorithms given above.

1.6.2.4 *Core locality*

The above concepts of spatial and temporal locality were mostly properties of programs, although hardware properties such as cacheline length and cache size play a role in analyzing the amount of locality. There is a third type of locality that is more intimately tied to hardware: *core locality*. This comes into play with multicore, multi-threaded programs.

A code's execution is said to exhibit core locality if write accesses that are spatially or temporally close are performed on the same core or processing unit. Core locality is not just a property of a program, but also to a large extent of how the program is executed in parallel.

The following issues are at play here.

- There are performance implications to *cache coherence* (section 1.4.1) if two cores both have a copy of a certain cacheline in their local stores. If they both read from it there is no problem. However, if one of them writes to it, the coherence protocol will copy the cacheline to the other core's local store. This takes up precious memory bandwidth, so it is to be avoided.

- Core locality will be affected if the OS is allowed do *thread migration*, thereby invalidating cache contents. Matters of fixing *thread affinity* are discussed in *Parallel Programming book, chapter 25* and *Parallel Programming book, chapter 45*.
- In multi-*socket* systems, there is also the phenomenon of *first touch*: data gets allocated on the socket where it is first initialized. This means that access from the other socket(s) may be considerably slower. See *Parallel Programming book, section 25.2*.

1.7 Further topics

1.7.1 Power consumption

Another important topic in high performance computers is their power consumption. Here we need to distinguish between the power consumption of a single processor chip, and that of a complete cluster.

As the number of components on a chip grows, its power consumption would also grow. Fortunately, in a counter acting trend, miniaturization of the chip features has simultaneously been reducing the necessary power. Suppose that the feature size λ (think: thickness of wires) is scaled down to $s\lambda$ with $s < 1$. In order to keep the electric field in the transistor constant, the length and width of the channel, the oxide thickness, substrate concentration density and the operating voltage are all scaled by the same factor.

1.7.1.1 Derivation of scaling properties

The properties of *constant field scaling* or *Dennard scaling* [18, 45] are an ideal-case description of the properties of a circuit as it is miniaturized. One important result is that power density stays constant as chip features get smaller, and the frequency is simultaneously increased.

The basic properties derived from circuit theory are that, if we scale feature size down by s:

Feature size	$\sim s$
Voltage	$\sim s$
Current	$\sim s$
Frequency	$\sim s$

Then we can derive that

$$\text{Power} = V \cdot I \sim s^2,$$

and because the total size of the circuit also goes down with s^2, the power density stays the same. Thus, it also becomes possible to put more transistors on a circuit, and essentially not change the cooling problem.

This result can be considered the driving force behind *Moore's law*, which states that the number of transistors in a processor doubles every 18 months.

Year	#transistors	Processor
1975	3,000	6502
1979	30,000	8088
1985	300,000	386
1989	1,000,000	486
1995	6,000,000	Pentium Pro
2000	40,000,000	Pentium 4
2005	100,000,000	2-core Pentium D
2008	700,000,000	8-core Nehalem

The frequency-dependent part of the power a processor needs comes from charging and discharging the capacitance of the circuit, so

$$\begin{array}{ll} \text{Charge} & q = CV \\ \text{Work} & W = qV = CV^2 \\ \text{Power} & W/\text{time} = WF = CV^2F \end{array} \tag{1.1}$$

This analysis can be used to justify the introduction of multicore processors.

1.7.1.2 Multicore

At the time of this writing (circa 2010), miniaturization of components has almost come to a standstill, because further lowering of the voltage would give prohibitive leakage. Conversely, the frequency can not be scaled up since this would raise the heat production of the chip too far. Figure 1.21 gives a dramatic illustration of the heat that a chip would give off, if single-

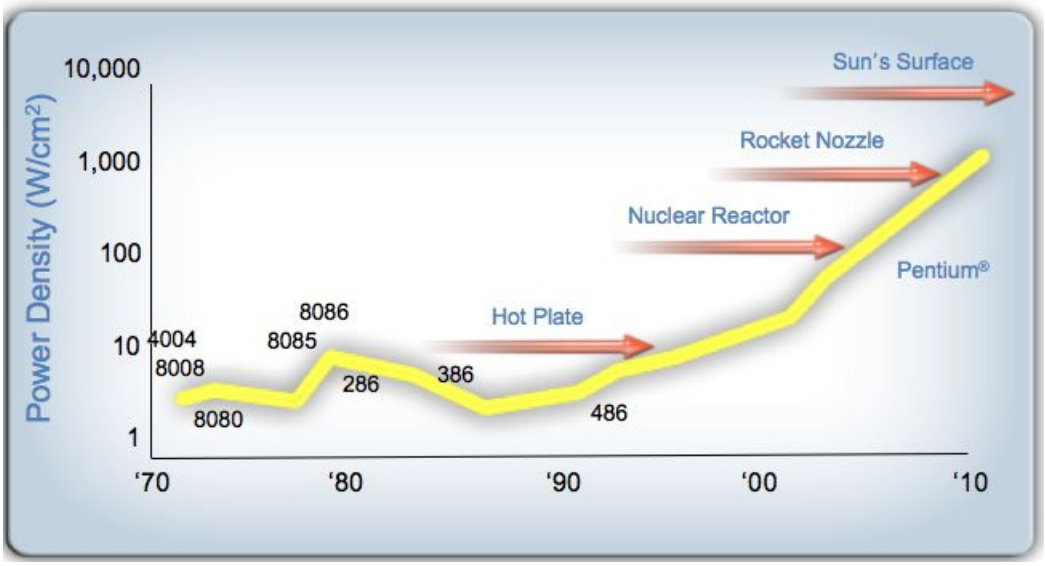


Figure 1.21: Projected heat dissipation of a CPU if trends had continued – this graph courtesy Pat Helsinger.

processor trends had continued.

One conclusion is that computer design is running into a *power wall*, where the sophistication of a single core can not be increased any further (so we can for instance no longer increase *ILP* and *pipeline depth*) and the only way to increase performance is to increase the amount of explicitly visible parallelism. This development has led to the current generation of *multicore* processors; see section 1.4. It is also the reason

GPUs with their simplified processor design and hence lower energy consumption are attractive; the same holds for FPGAs. One solution to the power wall problem is introduction of *multicore* processors. Recall equation 1.1, and compare a single processor to two processors at half the frequency. That should have the same computing power, right? Since we lowered the frequency, we can lower the voltage if we stay with the same process technology.

The total electric power for the two processors (cores) is, ideally,

$$\left.\begin{array}{l} C_{\text{multi}} = 2C \\ F_{\text{multi}} = F/2 \\ V_{\text{multi}} = V/2 \end{array}\right\} \Rightarrow P_{\text{multi}} = P/4.$$

In practice the capacitance will go up by a little over 2, and the voltage can not quite be dropped by 2, so it is more likely that $P_{\text{multi}} \approx 0.4 \times P$ [30]. Of course the integration aspects are a little more complicated in practice [19]; the important conclusion is that now, in order to lower the power (or, conversely, to allow further increase in performance while keeping the power constant) we now have to start programming in parallel.

1.7.1.3 Total computer power

The total power consumption of a parallel computer is determined by the consumption per processor and the number of processors in the full machine. At present, this is commonly several Megawatts. By the above reasoning, the increase in power needed from increasing the number of processors can no longer be offset by more power-effective processors, so power is becoming the overriding consideration as parallel computers move from the petascale (attained in 2008 by the *IBM Roadrunner*) to a projected exascale.

In the most recent generations of processors, power is becoming an overriding consideration, with influence in unlikely places. For instance, the Single Instruction Multiple Data (SIMD) design of processors (see section 2.3.1, in particular section 2.3.1.2) is dictated by the power cost of instruction decoding.

1.7.2 Operating system effects

HPC practitioners typically don't worry much about the *OS*. However, sometimes the presence of the OS can be felt, influencing performance. The reason for this is the *periodic interrupt*, where the operating system upwards of 100 times per second interrupts the current process to let another process or a system *daemon* have a *time slice*.

If you are running basically one program, you don't want the overhead and *jitter*, the unpredictability of process runtimes, this introduces. Therefore, computers have existed that basically dispensed with having an OS to increase performance.

The *periodic interrupt* has further negative effects. For instance, it pollutes the cache and TLB. As a fine-grained effect of jitter, it degrades performance of codes that rely on barriers between threads, such as frequently happens in OpenMP (section 2.6.2).

In particular in *financial applications*, where very tight synchronization is important, have adopted a Linux kernel mode where the periodic timer ticks only once a second, rather than hundreds of times. This is called a *tickless kernel*.

1.8 Review questions

For the true/false questions, give short explanation if you choose the 'false' answer.

Exercise 1.19. True or false. The code

```
for (int i=0; i<N; ++i)
  a[i] = b[i]+1;
```

touches every element of `a` and `b` once, so there will be a cache miss for each element.

Exercise 1.20. Give an example of a code fragment where a 3-way associative cache will have conflicts, but a 4-way cache will not.

Exercise 1.21. Consider the matrix-vector product with an $N \times N$ matrix. What is the needed cache size to execute this operation with only compulsory cache misses? Your answer depends on how the operation is implemented: answer separately for rowwise and columnwise traversal of the matrix, where you can assume that the matrix is always stored by rows.

Chapter 2

Parallel Computing

The largest and most powerful computers are sometimes called 'supercomputers'. For the last two decades, this has, without exception, referred to parallel computers: machines with more than one CPU that can be set to work on the same problem.

Parallelism is hard to define precisely, since it can appear on several levels. In the previous chapter you already saw how inside a CPU several instructions can be 'in flight' simultaneously. This is called *instruction-level parallelism*, and it is outside explicit user control: it derives from the compiler and the CPU deciding which instructions, out of a single instruction stream, can be processed simultaneously. At the other extreme is the sort of parallelism where more than one instruction stream is handled by multiple processors, often each on their own circuit board. This type of parallelism is typically explicitly scheduled by the user.

In this chapter, we will analyze this more explicit type of parallelism, the hardware that supports it, the programming that enables it, and the concepts that analyze it.

2.1 Introduction

In scientific codes, there is often a large amount of work to be done, and it is often regular to some extent, with the same operation being performed on many data. The question is then whether this work can be sped up by use of a parallel computer. If there are n operations to be done, and they would take time t on a single processor, can they be done in time t/p on p processors?

Let us start with a very simple example. Adding two vectors of length n

```
for (i=0; i<n; i++)
  a[i] = b[i] + c[i];
```

can be done with up to n processors. In the idealized case with n processors, each processor has local scalars `a,b,c` and executes the single instruction `a=b+c`. This is depicted in figure 2.1.

In the general case, where each processor executes something like

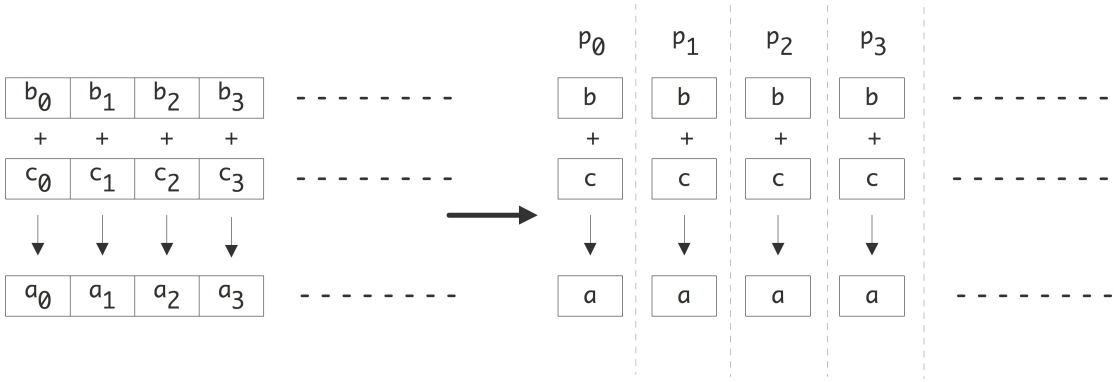


Figure 2.1: Parallelization of a vector addition.

```
for (i=my_low; i<my_high; i++)
  a[i] = b[i] + c[i];
```

execution time is linearly reduced with the number of processors. If each operation takes a unit time, the original algorithm takes time n, and the parallel execution on p processors n/p. The parallel algorithm is faster by a factor of p[1] .

Next, let us consider summing the elements of a vector. (An operation that has a vector as input but only a scalar as output is often called a *reduction*.) We again assume that each processor contains just a single array element. The sequential code:

```
s = 0;
for (i=0; i<n; i++)
  s += x[i]
```

is no longer obviously parallel, but if we recode the loop as

```
for (s=2; s<2*n; s*=2)
  for (i=0; i<n-s/2; i+=s)
    x[i] += x[i+s/2]
```

there is a way to parallelize it: every iteration of the outer loop is now a loop that can be done by n/s processors in parallel. Since the outer loop will go through $\log_2 n$ iterations, we see that the new algorithm has a reduced runtime of $n/p \cdot \log_2 n$. The parallel algorithm is now faster by a factor of $p/\log_2 n$. This is depicted in figure 2.2.

Even from these two simple examples we can see some of the characteristics of parallel computing:

- Sometimes algorithms need to be rewritten slightly to make them parallel.
- A parallel algorithm may not show perfect speedup.

1. Here we ignore lower order errors in this result when p does not divide perfectly in n. We will also, in general, ignore matters of loop overhead.

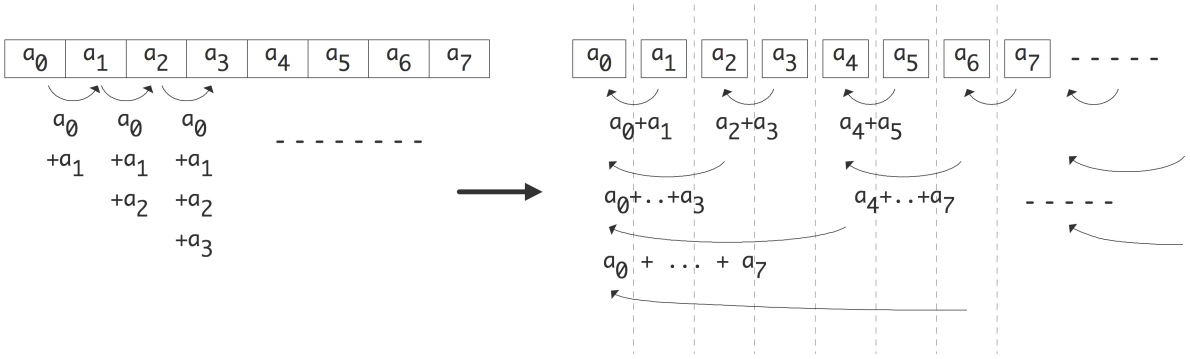


Figure 2.2: Parallelization of a vector reduction.

There are other things to remark on. In the first case, if each processors has its x_i, y_i in a local store the algorithm can be executed without further complications. In the second case, processors need to *communicate* data among each other and we haven't assigned a cost to that yet.

First let us look systematically at communication. We can take the parallel algorithm in the right half of figure 2.2 and turn it into a tree graph (see Appendix 20) by defining the inputs as leave nodes, all partial sums as interior nodes, and the root as the total sum. There is an edge from one node to another if the first is input to the (partial) sum in the other. This is illustrated in figure 2.3. In this figure nodes are horizontally aligned with other computations that can be performed simultaneously; each level is sometimes called a *superstep* in the computation. Nodes are vertically aligned if they are computed on the same processors, and an arrow corresponds to a communication if it goes from one processor to another. The vertical alignment in figure 2.3 is not the only one possible. If nodes are shuffled within a superstep

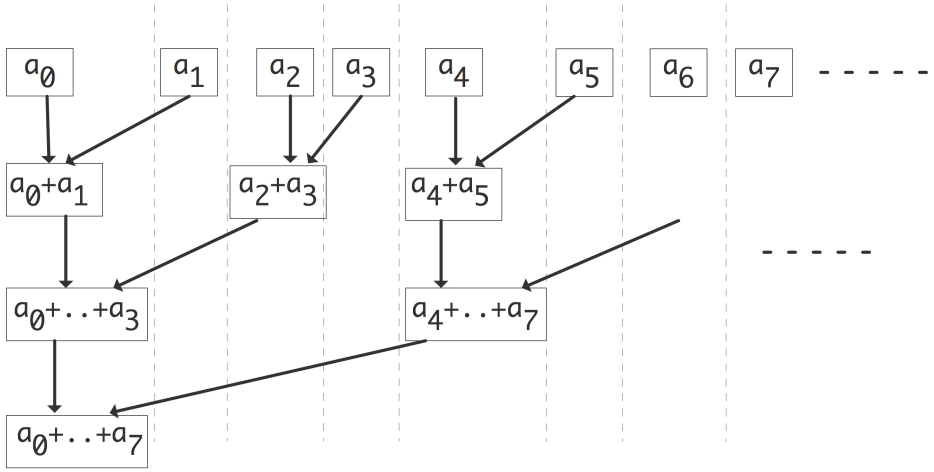


Figure 2.3: Communication structure of a parallel vector reduction.

or horizontal level, a different communication pattern arises.

Exercise 2.1. Consider placing the nodes within a superstep on random processors. Show

that, if no two nodes wind up on the same processor, at most twice the number of communications is performed from the case in figure 2.3.

Exercise 2.2. Can you draw the graph of a computation that leaves the sum result on each processor? There is a solution that takes twice the number of supersteps, and there is one that takes the same number. In both cases the graph is no longer a tree, but a more general Directed Acyclic Graph (DAG).

Processors are often connected through a network, and moving data through this network takes time. This introduces a concept of distance between the processors. In figure 2.3, where the processors are linearly ordered, this is related to their rank in the ordering. If the network only connects a processor with its immediate neighbors, each iteration of the outer loop increases the distance over which communication takes place.

Exercise 2.3. Assume that an addition takes a certain unit time, and that moving a number from one processor to another takes that same unit time. Show that the communication time equals the computation time.
Now assume that sending a number from processor p to $p \pm k$ takes time k. Show that the execution time of the parallel algorithm now is of the same order as the sequential time.

The summing example made the unrealistic assumption that every processor initially stored just one vector element: in practice we will have $p < n$, and every processor stores a number of vector elements. The obvious strategy is to give each processor a consecutive stretch of elements, but sometimes the obvious strategy is not the best.

Exercise 2.4. Consider the case of summing 8 elements with 4 processors. Show that some of the edges in the graph of figure 2.3 no longer correspond to actual communications.
Now consider summing 16 elements with, again, 4 processors. What is the number of communication edges this time?

These matters of algorithm adaptation, efficiency, and communication, are crucial to all of parallel computing. We will return to these issues in various guises throughout this chapter.

2.1.1 Functional parallelism versus data parallelism

From the above introduction we can describe parallelism as finding independent operations in the execution of a program. In all of the examples these independent operations were in fact identical operations, but applied to different data items. We could call this *data parallelism*: the same operation is applied in parallel to many data elements. This is in fact a common scenario in scientific computing: parallelism often stems from the fact that a data set (vector, matrix, graph,...) is spread over many processors, each working on its part of the data.

The term data parallelism is traditionally mostly applied if the operation is a single instruction; in the case of a subprogram it is often called *task parallelism*.

It is also possible to find independence, not based on data elements, but based on the instructions themselves. Traditionally, compilers analyze code in terms of ILP: independent instructions can be given to separate floating point units, or reordered, for instance to optimize register usage (see also section 2.5.2).

ILP is one case of *functional parallelism*; on a higher level, functional parallelism can be obtained by considering independent subprograms, often called *task parallelism*; see section 2.5.3.

Some examples of functional parallelism are Monte Carlo simulations, and other algorithms that traverse a parametrized search space, such as boolean *satisfyability* problems.

2.1.2 Parallelism in the algorithm versus in the code

Often we are in the situation that we want to parallelize an algorithm that has a common expression in sequential form. In some cases, this sequential form is straightforward to parallelize, such as in the vector addition discussed above. In other cases there is no simple way to parallelize the algorithm; we will discuss linear recurrences in section 7.10.2. And in yet another case the sequential code may look not parallel, but the algorithm actually has parallelism.

Exercise 2.5.

```
for i in [1:N]:
    x[0,i] = some_function_of(i)
    x[i,0] = some_function_of(i)

for i in [1:N]:
    for j in [1:N]:
        x[i,j] = x[i-1,j]+x[i,j-1]
```

Answer the following questions about the double `i,j` loop:

1. Are the iterations of the inner loop independent, that is, could they be executed simultaneously?
2. Are the iterations of the outer loop independent?
3. If `x[1,1]` is known, show that `x[2,1]` and `x[1,2]` can be computed independently.
4. Does this give you an idea for a parallelization strategy?

We will discuss the solution to this conundrum in section 7.10.1. In general, the whole of chapter 7 will be about the amount of parallelism intrinsic in scientific computing algorithms.

2.2 Theoretical concepts

There are two important reasons for using a parallel computer: to have access to more memory or to obtain higher performance. It is easy to characterize the gain in memory, as the total memory is the sum of the individual memories. The speed of a parallel computer is harder to characterize. This section will have an extended discussion on theoretical measures for expressing and judging the gain in execution speed from going to a parallel architecture.

2.2.1 Definitions

2.2.1.1 *Speedup and efficiency*

A simple approach to defining speedup is to let the same program run on a single processor, and on a parallel machine with p processors, and to compare runtimes. With T_1 the execution time on a single processor and T_p the time on p processors, we define the *speedup* as $S_p = T_1/T_p$. (Sometimes T_1 is defined as 'the best time to solve the problem on a single processor', which allows for using a different algorithm on a single processor than in parallel.) In the ideal case, $T_p = T_1/p$, but in practice we don't expect to attain that, so $S_p \leq p$. To measure how far we are from the ideal speedup, we introduce the *efficiency* $E_p = S_p/p$. Clearly, $0 < E_p \leq 1$.

Exercise 2.6. Prove that $E = 1$ implies that all processors are active all the time.
(Hint: suppose all processors finish their work in time T, except for one processor in $T' < T$. What is T_p in this case? Explore the above relations.)

There is a practical problem with the above definitions: a problem that can be solved on a parallel machine may be too large to fit on any single processor. Conversely, distributing a single processor problem over many processors may give a distorted picture since very little data will wind up on each processor. Below we will discuss more realistic measures of speed-up.

There are various reasons why the actual speed is less than p. For one, using more than one processor necessitates communication and synchronization, which is overhead that was not part of the original computation. Secondly, if the processors do not have exactly the same amount of work to do, they may be idle part of the time (this is known as *load unbalance*), again lowering the actually attained speedup. Finally, code may have sections that are inherently sequential.

Communication between processors is an important source of a loss of efficiency. Clearly, a problem that can be solved without communication will be very efficient. Such problems, in effect consisting of a number of completely independent calculations, is called *embarrassingly parallel* (or *conveniently parallel*; see section 2.5.4); it will have close to a perfect speedup and efficiency.

Exercise 2.7. The case of speedup larger than the number of processors is called *superlinear speedup*. Give a theoretical argument why this can never happen.

In practice, superlinear speedup can happen. For instance, suppose a problem is too large to fit in memory, and a single processor can only solve it by swapping data to disc. If the same problem fits in the memory of two processors, the speedup may well be larger than 2 since disc swapping no longer occurs. Having less, or more localized, data may also improve the cache behavior of a code.

A form of superlinear speedup can also happen in *search* algorithms. Imagine that each processor starts in a different location of the search space; now it can happen that, say, processor 3 immediately finds the solution. Sequentially, all the possibilities for processors 1 and 2 would have had to be traversed. The speedup is much greater than 3 in this case.

2.2.1.2 *Cost-optimality*

In cases where the speedup is not perfect we can define *overhead* as the difference

$$T_o = pT_p - T_1.$$

We can also interpret this as the difference between simulating the parallel algorithm on a single processor, and the actual best sequential algorithm.

We will later see two different types of overhead:

1. The parallel algorithm can be essentially different from the sequential one. For instance, sorting algorithms have a complexity $O(n \log n)$, but the parallel bitonic sort (section 9.6) has complexity $O(n \log^2 n)$.
2. The parallel algorithm can have overhead derived from the process or parallelizing, such as the cost of sending messages. As an example, section 7.2.3 analyzes the communication overhead in the matrix-vector product.

A parallel algorithm is called *cost-optimal* if the overhead is at most of the order of the running time of the sequential algorithm.

Exercise 2.8. The definition of overhead above implicitly assumes that overhead is not parallelizable. Discuss this assumption in the context of the two examples above.

2.2.2 Asymptotics

If we ignore limitations such as that the number of processors has to be finite, or the physicalities of the interconnect between them, we can derive theoretical results on the limits of parallel computing. This section will give a brief introduction to such results, and discuss their connection to real life high performance computing.

Consider for instance the matrix-matrix multiplication $C = AB$, which takes $2N^3$ operations where N is the matrix size. Since there are no dependencies between the operations for the elements of C, we can perform them all in parallel. If we had N^2 processors, we could assign each to an (i, j) coordinate in C, and have it compute c_{ij} in $2N$ time. Thus, this parallel operation has efficiency 1, which is optimal.

Exercise 2.9. Show that this algorithm ignores some serious issues about memory usage:

- If the matrix is kept in shared memory, how many simultaneous reads from each memory locations are performed?
- If the processors keep the input and output to the local computations in local storage, how much duplication is there of the matrix elements?

Adding N numbers $\{x_i\}_{i=1\ldots N}$ can be performed in $\log_2 N$ time with $N/2$ processors. If we have $N/2$ processors we could compute:

1. Define $s_i^{(0)} = x_i$.
2. Iterate with $j = 1, \ldots, \log_2 N$:
3. Compute $N/2^j$ partial sums $s_i^{(j)} = s_{2i}^{(j-1)} + s_{2i+1}^{(j-1)}$

We see that the $N/2$ processors perform a total of N operations (as they should) in $\log_2 N$ time. The efficiency of this parallel scheme is $O(1/\log_2 N)$, a slowly decreasing function of N.

Exercise 2.10. Show that, with the scheme for parallel addition just outlined, you can multiply two matrices in $\log_2 N$ time with $N^3/2$ processors. What is the resulting efficiency?

It is now a legitimate theoretical question to ask

- If we had infinitely many processors, what is the lowest possible time complexity for matrix-matrix multiplication, or
- Are there faster algorithms that still have $O(1)$ efficiency?

Such questions have been researched (see for instance [98]), but they have little bearing on high performance computing.

A first objection to these kinds of theoretical bounds is that they implicitly assume some form of shared memory. In fact, the formal model for these algorithms is called a *Parallel Random Access Machine (PRAM)*, where the assumption is that every memory location is accessible to any processor.

Often an additional assumption is made that multiple simultaneous accesses to the same location are in fact possible. Since write and write accesses have a different behavior in practice, there is the concept of CREW-PRAM, for Concurrent Read, Exclusive Write PRAM.

The basic assumptions of the PRAM model are unrealistic in practice, especially in the context of scaling up the problem size and the number of processors. A further objection to the PRAM model is that even on a single processor it ignores the memory hierarchy; section 1.3.

But even if we take distributed memory into account, theoretical results can still be unrealistic. The above summation algorithm can indeed work unchanged in distributed memory, except that we have to worry about the distance between active processors increasing as we iterate further. If the processors are connected by a linear array, the number of 'hops' between active processors doubles, and with that, asymptotically, the computation time of the iteration. The total execution time then becomes $n/2$, a disappointing result given that we throw so many processors at the problem.

What if the processors are connected with a hypercube topology (section 2.7.5)? It is not hard to see that the summation algorithm can then indeed work in $\log_2 n$ time. However, as $n \to \infty$, can we physically construct a sequence of hypercubes of n nodes and keep the communication time between two connected constant? Since communication time depends on latency, which partly depends on the length of the wires, we have to worry about the physical distance between nearest neighbors.

The crucial question here is whether the hypercube (an n-dimensional object) can be embedded in 3-dimensional space, while keeping the distance (measured in meters) constant between connected neighbors. It is easy to see that a 3-dimensional grid can be scaled up arbitrarily while maintaining a unit wire length, but the question is not clear for a hypercube. There, the length of the wires may have to increase as n grows, which runs afoul of the finite speed of electrons.

We sketch a proof (see [63] for more details) that, in our three dimensional world and with a finite speed of light, speedup is limited to $\sqrt[4]{n}$ for a problem on n processors, no matter the interconnect. The argument goes as follows. Consider an operation that involves collecting a final result on one processor. Assume that each processor takes a unit volume of space, produces one result per unit time, and can send one data item per unit time. Then, in an amount of time t, at most the processors in a ball with radius t, that is, $O(t^3)$ processors total, can contribute to the final result; all others are too far away. In time T, then, the number of operations that can contribute to the final result is $\int_0^T t^3 dt = O(T^4)$. This means that the maximum achievable speedup is the fourth root of the sequential time.

Finally, the question 'what if we had infinitely many processors' is not realistic as such, but we will allow it in the sense that we will ask the *weak scaling* question (section 2.2.5) 'what if we let the problem size and

the number of processors grow proportional to each other'. This question is legitimate, since it corresponds to the very practical deliberation whether buying more processors will allow one to run larger problems, and if so, with what 'bang for the buck'.

2.2.3 Amdahl's law

One reason for less than perfect speedup is that parts of a code can be inherently sequential. This limits the parallel efficiency as follows. Suppose that 5% of a code is sequential, then the time for that part can not be reduced, no matter how many processors are available. Thus, the speedup on that code is limited to a factor of 20. This phenomenon is known as *Amdahl's Law* [3], which we will now formulate.

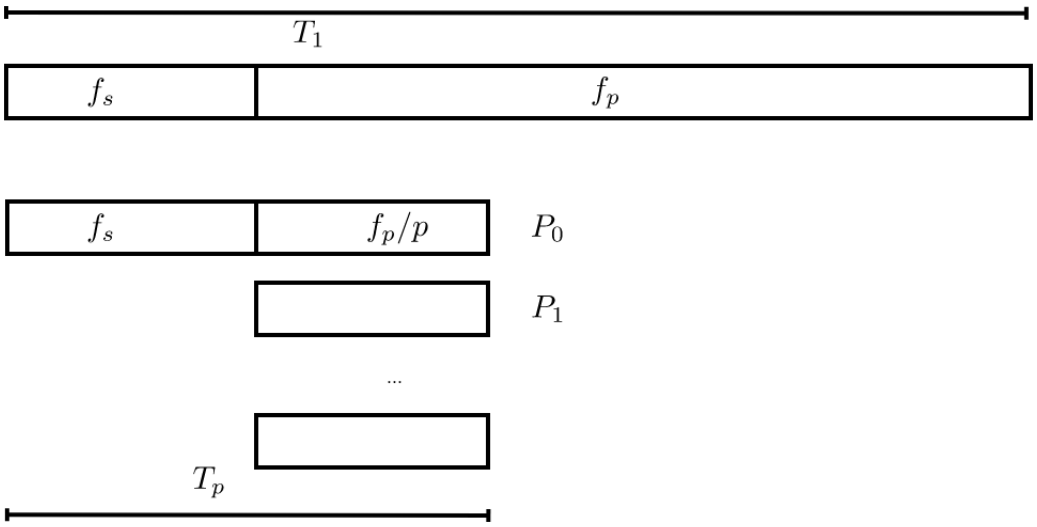


Figure 2.4: Sequential and parallel time in Amdahl's analysis.

Let F_s be the *sequential fraction* and F_p be the *parallel fraction* (or more strictly: the 'parallelizable' fraction) of a code, respectively. Then $F_p + F_s = 1$. The parallel execution time T_p on p processors is the sum of the part that is sequential $T_1 F_s$ and the part that can be parallelized $T_1 F_p / P$:

$$T_P = T_1(F_s + F_p/P). \tag{2.1}$$

(see figure 2.4) As the number of processors grows $P \to \infty$, the parallel execution time now approaches that of the sequential fraction of the code: $T_P \downarrow T_1 F_s$. We conclude that speedup is limited by $S_P \leq 1/F_s$ and efficiency is a decreasing function $E \sim 1/P$.

The sequential fraction of a code can consist of things such as I/O operations. However, there are also parts of a code that in effect act as sequential. Consider a program that executes a single loop, where all iterations can be computed independently. Clearly, this code offers no obstacles to parallelization. However by splitting the loop in a number of parts, one per processor, each processor now has to deal with loop overhead: calculation of bounds, and the test for completion. This overhead is replicated as many times as there are processors. In effect, loop overhead acts as a sequential part of the code.

Exercise 2.11. Let's do a specific example. Assume that a code has a setup that takes 1 second and a parallelizable section that takes 1000 seconds on one processor. What are the speedup and efficiency if the code is executed with 100 processors? What are they for 500 processors? Express your answer to at most two significant digits.

Exercise 2.12. Investigate the implications of Amdahl's law: if the number of processors P increases, how does the parallel fraction of a code have to increase to maintain a fixed efficiency?

2.2.3.1 *Amdahl's law with communication overhead*

In a way, Amdahl's law, sobering as it is, is even optimistic. Parallelizing a code will give a certain speedup, but it also introduces *communication overhead* that will lower the speedup attained. Let us refine our model of equation (2.1) (see [132, p. 367]):

$$T_p = T_1(F_s + F_p/P) + T_c,$$

where T_c is a fixed communication time.

To assess the influence of this communication overhead, we assume that the code is fully parallelizable, that is, $F_p = 1$. We then find that

$$S_p = \frac{T_1}{T_1/p + T_c}. \tag{2.2}$$

For this to be close to p, we need $T_c \ll T_1/p$ or $p \ll T_1/T_c$. In other words, the number of processors should not grow beyond the ratio of scalar execution time and communication overhead.

2.2.3.2 *Gustafson's law*

Amdahl's law was thought to show that large numbers of processors would never pay off. However, the implicit assumption in Amdahl's law is that there is a fixed computation which gets executed on more and more processors. In practice this is not the case: typically there is a way of *scaling* up a *problem* (in chapter 4 you will learn the concept of 'discretization'), and one tailors the size of the problem to the number of available processors.

A more realistic assumption would be to say that there is a sequential fraction independent of the problem size, and a parallel fraction that can be arbitrarily replicated. To formalize this, instead of starting with the execution time of the sequential program, let us start with the execution time of the parallel program, and say that

$$T_p = T(F_s + F_p) \qquad \text{with } F_s + F_p = 1.$$

Now we have two possible definitions of T_1. First of all, there is the T_1 you get from setting $p = 1$ in T_p. (Convince yourself that that is actually the same as T_p.) However, what we need is T_1 describing the time to do all the operations of the parallel program. (See figure 2.5.) Thus, we start out with an observed T_p, and reconstruct T_1 as:

$$T_1 = F_s T_p + p \cdot F_p T_p.$$

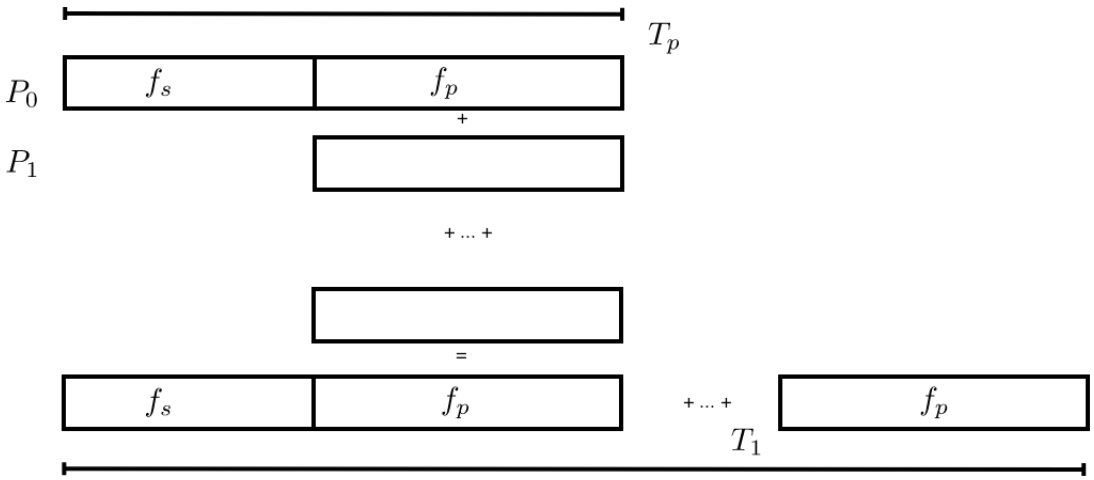


Figure 2.5: Sequential and parallel time in Gustafson's analysis.

This gives us a speedup of

$$S_p = \frac{T_1}{T_p} = \frac{F_s + p \cdot F_p}{F_s + F_p} = F_s + p \cdot F_p = p - (p-1) \cdot F_s. \tag{2.3}$$

From this formula we see that:

- Speedup is still bounded by p;
- ... but it's a positive number;
- for a given p it's again a decreasing function of the sequential fraction.

Exercise 2.13. Rewrite equation (2.3) to express the speedup in terms of p and F_p. What is the asymptotic behavior of the efficiency E_p?

As with Amdahl's law, we can investigate the behavior of Gustafson's law if we include communication overhead. Let's go back to equation (2.2) for a perfectly parallelizable problem, and approximate it as

$$S_p = p(1 - \frac{T_c}{T_1}p).$$

Now, under the assumption of a problem that is gradually being scaled up, T_c and T_1 become functions of p. We see that, if $T_1(p) \sim pT_c(p)$, we get linear speedup that is a constant fraction away from 1. As a general discussion we can not take this analysis further; in section 7.2.3 you'll see a detailed analysis of an example.

2.2.3.3 *Amdahl's law and hybrid programming*

Above, you learned about hybrid programming, a mix between distributed and shared memory programming. This leads to a new form of Amdahl's law.

Suppose we have p nodes with c cores each, and F_p describes the fraction of the code that uses c-way thread parallelism. We assume that the whole code is fully parallel over the p nodes. The ideal speed up

would be pc, and the ideal parallel running time $T_1/(pc)$, but the actual running time is

$$T_{p,c} = T_1 \left(\frac{F_s}{p} + \frac{F_p}{pc} \right) = \frac{T_1}{pc} \left(F_s c + F_p \right) = \frac{T_1}{pc} \left(1 + F_s(c-1) \right).$$

Exercise 2.14. Show that the speedup $T_1/T_{p,c}$ can be approximated by p/F_s.

In the original Amdahl's law, speedup was limited by the sequential portion to a fixed number $1/F_s$, in hybrid programming it is limited by the task parallel portion to p/F_s.

2.2.4 Critical path and Brent's theorem

The above definitions of speedup and efficiency, and the discussion of Amdahl's law and Gustafson's law, made an implicit assumption that parallel work can be arbitrarily subdivided. As you saw in the summing example in section 2.1, this may not always be the case: there can be dependencies between operations, meaning that one operation depends on an earlier in the sense of needing its result as input. Dependent operations can not be executed simultaneously, so they limit the amount of parallelism that can be employed.

We define the *critical path* as a (possibly non-unique) chain of dependencies of maximum length. (This length is sometimes known as *span*.) Since the tasks on a critical path need to be executed one after another, the length of the critical path is a lower bound on parallel execution time.

To make these notions precise, we define the following concepts:

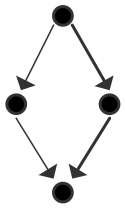
Definition 1

T_1 : *the time the computation takes on a single processor*
T_p : *the time the computation takes with p processors*
T_∞ : *the time the computation takes if unlimited processors are available*
P_∞ : *the value of p for which* $T_p = T_\infty$

With these concepts, we can define the *average parallelism* of an algorithm as T_1/T_∞, and the length of the critical path is T_∞.

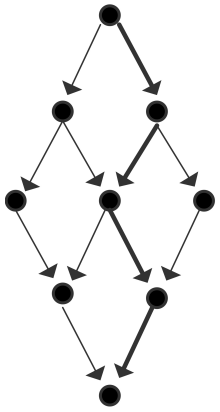
Exercise 2.15. Suppose you have an algorithm that can be executed with $E_p \equiv 1$. What does that mean for the critical path analysis?

We will now give a few illustrations by showing a graph of tasks and their dependencies. We assume for simplicity that each node is a unit time task.



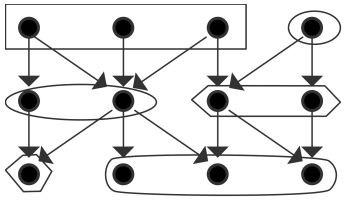
The maximum number of processors that can be used is 2 and the average parallelism is 4/3:

$$T_1 = 4, \quad T_\infty = 3 \quad \Rightarrow T_1/T_\infty = 4/3$$
$$T_2 = 3, \quad S_2 = 4/3, \quad E_2 = 2/3$$
$$P_\infty = 2$$



The maximum number of processors that can be used is 3 and the average parallelism is $9/5$; efficiency is maximal for $p = 2$:

$$\begin{array}{llll} T_1 = 9, & T_\infty = 5 & \Rightarrow T_1/T_\infty = 9/5 \\ T_2 = 6, & S_2 = 3/2, & E_2 = 3/4 \\ T_3 = 5, & S_3 = 9/5, & E_3 = 3/5 \\ P_\infty = 3 \end{array}$$



The maximum number of processors that can be used is 4 and that is also the average parallelism; the figure illustrates a parallelization with $P = 3$ that has efficiency $\equiv 1$:

$$\begin{array}{llll} T_1 = 12, & T_\infty = 4 & \Rightarrow T_1/T_\infty = 3 \\ T_2 = 6, & S_2 = 2, & E_2 = 1 \\ T_3 = 4, & S_3 = 3, & E_3 = 1 \\ T_4 = 3, & S_4 = 4, & E_4 = 1 \\ P_\infty = 4 \end{array}$$

Based on these examples, you probably see that there are two extreme cases:

- If every task depends on precisely on other, you get a chain of dependencies, and $T_p = T_1$ for any p.
- On the other hand, if all tasks are independent (and p divides their number) you get $T_p = T_1/p$ for any p.
- In a slightly less trivial scenario than the previous, consider the case where the critical path is of length m, and in each of these m steps there are $p-1$ independent tasks, or at least: dependent only on tasks in the previous step. There will then be perfect parallelism in each of the m steps, and we can express $T_p = T_1/p$ or $T_p = m + (T_1 - m)/p$.

That last statement actually holds in general. This is known as *Brent's theorem*:

Theorem 1 *Let m be the total number of tasks, p the number of processors, and t the length of a* critical path. *Then the computation can be done in*

$$T_p \leq t + \frac{m-t}{p}.$$

Proof. Divide the computation in steps, such that tasks in step $i+1$ are independent of each other, and only dependent on step i. Let s_i be the number of tasks in step i, then the time for that step is $\lceil \frac{s_i}{p} \rceil$. Summing over i gives

$$T_p = \sum_i^t \lceil \frac{s_i}{p} \rceil \leq \sum_i^t \frac{s_i + p - 1}{p} = t + \sum_i^t \frac{s_i - 1}{p} = t + \frac{m-t}{p}.$$

Exercise 2.16. Consider a tree of depth d, that is, with $2^d - 1$ nodes, and a search

$$\max_{n \in \text{nodes}} f(n).$$

Assume that all nodes need to be visited: we have no knowledge or any ordering on their values.
Analyze the parallel running time on p processors, where you may assume that $p = 2^q$, with $q < d$. How does this relate to the numbers you get from Brent's theorem and Amdahl's law?

2.2.5 Scalability

Above, we remarked that splitting a given problem over more and more processors does not make sense: at a certain point there is just not enough work for each processor to operate efficiently. Instead, in practice users of a parallel code will either choose the number of processors to match the problem size, or they will solve a series of increasingly larger problems on correspondingly growing numbers of processors. In both cases it is hard to talk about speedup. Instead, the concept of *scalability* is used.

We distinguish two types of scalability. So-called *strong scalability* is in effect the same as speedup as discussed above. We say that a problem shows strong scalability if, partitioned over more and more processors, it shows perfect or near perfect speedup, that is, the execution time goes down linearly with the number of processors. In terms of efficiency we can describe this as:

$$\left.\begin{array}{l} N \equiv \text{constant} \\ P \to \infty \end{array}\right\} \Rightarrow E_P \approx \text{constant}$$

Typically, one encounters statements like 'this problem scales up to 500 processors', meaning that up to 500 processors the speedup will not noticeably decrease from optimal. It is not necessary for this problem to fit on a single processor: often a smaller number such as 64 processors is used as the baseline from which scalability is judged.

Exercise 2.17. We can formulate strong scaling as a runtime that is inversely proportional to the number of processors:

$$t = c/p.$$

Show that on a log-log plot, that is, you plot the logarithm of the runtime against the logarithm of the number of processors, you will get a straight line with slope -1.
Can you suggest a way of dealing with a non-parallelizable section, that is, with a runtime $t = c_1 + c_2/p$?

More interestingly, *weak scalability* describes the behavior of execution as problem size and number of processors both grow, but in such a way that the amount of work per processor stays constant. The term 'work' here is ambiguous: sometimes weak scaling is interpreted as keeping the amount of data constant, in other cases it's the number of operations that stays constant.

Measures such as speedup are somewhat hard to report, since the relation between the number of operations and the amount of data can be complicated. If this relation is linear, one could state that the amount

of data per processor is kept constant, and report that parallel execution time is constant as the number of processors grows. (Can you think of applications where the relation between work and data is linear? Where it is not?)

In terms of efficiency:

$$\left.\begin{array}{l} N \to \infty \\ P \to \infty \\ M = N/P \equiv \text{constant} \end{array}\right\} \Rightarrow E_P \approx \text{constant}$$

Exercise 2.18. Suppose you are investigating the weak scalability of a code. After running it for a couple of sizes and corresponding numbers of processes, you find that in each case the flop rate is roughly the same. Argue that the code is indeed weakly scalable.

Exercise 2.19. In the above discussion we always implicitly compared a sequential algorithm and the parallel form of that same algorithm. However, in section 2.2.1 we noted that sometimes speedup is defined as a comparison of a parallel algorithm with the **best** sequential algorithm for the same problem. With that in mind, compare a parallel sorting algorithm with runtime $(\log n)^2$ (for instance, *bitonic sort*; section 9.6) to the best serial algorithm, which has a running time of $n \log n$.
Show that in the weak scaling case of $n = p$ speedup is $p/\log p$. Show that in the strong scaling case speedup is a descending function of n.

Remark *A historical anecdote.*

```
Message:  1023110, 88 lines
Posted:  5:34pm EST, Mon Nov 25/85,
Subject:  Challenge from Alan Karp
To:  Numerical-Analysis, ...  From GOLUB@SU-SCORE.ARPA

I have just returned from the Second SIAM Conference on
Parallel Processing for Scientific Computing in Norfolk,
Virginia.  There I heard about 1,000 processor systems,
4,000 processor systems, and even a proposed 1,000,000
processor system.  Since I wonder if such systems are the
best way to do general purpose, scientific computing, I am
making the following offer.

I will pay $100 to the first person to demonstrate a speedup
of at least 200 on a general purpose, MIMD computer used for
scientific computing.  This offer will be withdrawn at 11:59
PM on 31 December 1995.
```

This was satisfied by scaling up the problem.

2.2.5.1 Iso-efficiency

In the definition of *weak scalability* above, we stated that, under some relation between problem size N and number of processors p, efficiency will stay constant. We can make this precise and define the *iso-*

efficiency curve as the relation between N, p that gives constant efficiency [83].

2.2.5.2 Precisely what do you mean by scalable?

In scientific computing scalability is a property of an algorithm and the way it is parallelized on an architecture, in particular noting the way data is distributed.

In computer *industry parlance* the term 'scalability' is sometimes applied to architectures or whole computer systems:

> A scalable computer is a computer designed from a small number of basic components, without a single bottleneck component, so that the computer can be incrementally expanded over its designed scaling range, delivering linear incremental performance for a well-defined set of scalable applications. General-purpose scalable computers provide a wide range of processing, memory size, and I/O resources. Scalability is the degree to which performance increments of a scalable computer are linear" [11].

In particular,

- *horizontal scaling* is adding more hardware components, such as adding nodes to a cluster;
- *vertical scaling* corresponds to using more powerful hardware, for instance by doing a hardware upgrade.

2.2.6 Simulation scaling

In most discussions of weak scaling we assume that the amount of work and the amount of storage are linearly related. This is not always the case; for instance the operation complexity of a matrix-matrix product is N^3 for N^2 data. If you linearly increase the number of processors, and keep the data per process constant, the work may go up with a higher power.

A similar effect comes into play if you simulate time-dependent PDEs. (This uses concepts from chapter 4.) Here, the total work is a product of the work per time step and the number of time steps. These two numbers are related; in section 4.1.2 you will see that the time step has a certain minimum size as a function of the space discretization. Thus, the number of time steps will go up as the work per time step goes up.

Consider now applications such as *weather prediction*, and say that currently you need 4 hours of compute time to predict the next 24 hours of weather. The question is now: if you are happy with these numbers, and you buy a bigger computer to get a more accurate prediction, what can you say about the new computer?

In other words, rather than investigating scalability from the point of the running of an algorithm, in this section we will look at the case where the simulated time S and the running time T are constant. Since the new computer is presumably faster, we can do more operations in the same amount of running time. However, it is not clear what will happen to the amount of data involved, and hence the memory required. To analyze this we have to use some knowledge of the math of the applications.

Let m be the memory per processor, and P the number of processors, giving:

$M = Pm$ total memory.

If d is the number of space dimensions of the problem, typically 2 or 3, we get

$$\Delta x = 1/M^{1/d} \qquad \text{grid spacing.}$$

For stability this limits the time step Δt to

$$\Delta t = \begin{cases} \Delta x = 1 \,/\, M^{1/d} & \text{hyperbolic case} \\ \Delta x^2 = 1 \,/\, M^{2/d} & \text{parabolic case} \end{cases}$$

(noting that the hyperbolic case was not discussed in chapter 4.) With a simulated time S, we find

$$k = S/\Delta t \qquad \text{time steps.}$$

If we assume that the individual time steps are perfectly parallelizable, that is, we use explicit methods, or implicit methods with optimal solvers, we find a running time

$$T = kM/P = \frac{S}{\Delta t} m.$$

Setting $T/S = C$, we find

$$m = C\Delta t,$$

that is, the amount of memory per processor goes down as we increase the processor count. (What is the missing step in that last sentence?)

Further analyzing this result, we find

$$m = C\Delta t = c \begin{cases} 1 \,/\, M^{1/d} & \text{hyperbolic case} \\ 1 \,/\, M^{2/d} & \text{parabolic case} \end{cases}$$

Substituting $M = Pm$, we find ultimately

$$m = C \begin{cases} 1 \,/\, P^{1/(d+1)} & \text{hyperbolic} \\ 1 \,/\, P^{2/(d+2)} & \text{parabolic} \end{cases}$$

that is, the memory per processor that we can use goes down as a higher power of the number of processors.

2.2.7 Other scaling measures

Amdahl's law above was formulated in terms of the execution time on one processor. In many practical situations this is unrealistic, since the problems executed in parallel would be too large to fit on any single processor. Some formula manipulation gives us quantities that are to an extent equivalent, but that do not rely on this single-processor number [152].

For starters, applying the definition $S_p(n) = \frac{T_1(n)}{T_p(n)}$ to strong scaling, we observe that $T_1(n)/n$ is the sequential time per operation, and its inverse $n/T_1(n)$ can be called the sequential *computational rate*, denoted $R_1(n)$. Similarly defining a 'parallel computational rate'

$$R_p(n) = n/T_p(n)$$

we find that

$$S_p(n) = R_p(n)/R_1(n)$$

In strong scaling $R_1(n)$ will be a constant, so we make a logarithmic plot of speedup, purely based on measuring $T_p(n)$.

2.2.8 Concurrency; asynchronous and distributed computing

Even on computers that are not parallel there is a question of the execution of multiple simultaneous processes. Operating systems typically have a concept of *time slicing*, where all active process are given command of the CPU for a small slice of time in rotation. In this way, a sequential can emulate a parallel machine; of course, without the efficiency.

However, time slicing is useful even when not running a parallel application: OSs will have independent processes (your editor, something monitoring your incoming mail, et cetera) that all need to stay active and run more or less often. The difficulty with such independent processes arises from the fact that they sometimes need access to the same resources. The situation where two processes both need the same two resources, each getting hold of one, is called *deadlock*. A famous formalization of *resource contention* is known as the *dining philosophers* problem.

The field that studies such as independent processes is variously known as *concurrency*, *asynchronous computing*, or *distributed computing*. The term concurrency describes that we are dealing with tasks that are simultaneously active, with no temporal ordering between their actions. The term distributed computing derives from such applications as database systems, where multiple independent clients need to access a shared database.

We will not discuss this topic much in this book. Section 2.6.1 discusses the *thread* mechanism that supports time slicing; on modern multicore processors threads can be used to implement shared memory parallel computing.

The book 'Communicating Sequential Processes' offers an analysis of the interaction between concurrent processes [106]. Other authors use topology to analyze asynchronous computing [101].

2.3 Parallel Computers Architectures

For quite a while now, the top computers have been some sort of parallel computer, that is, an architecture that allows the simultaneous execution of multiple instructions or instruction sequences. One way of characterizing the various forms this can take is due to Flynn [64]. *Flynn's taxonomy* characterizes architectures by whether the *data flow* and *control flow* are shared or independent. The following four types result (see also figure 2.6):

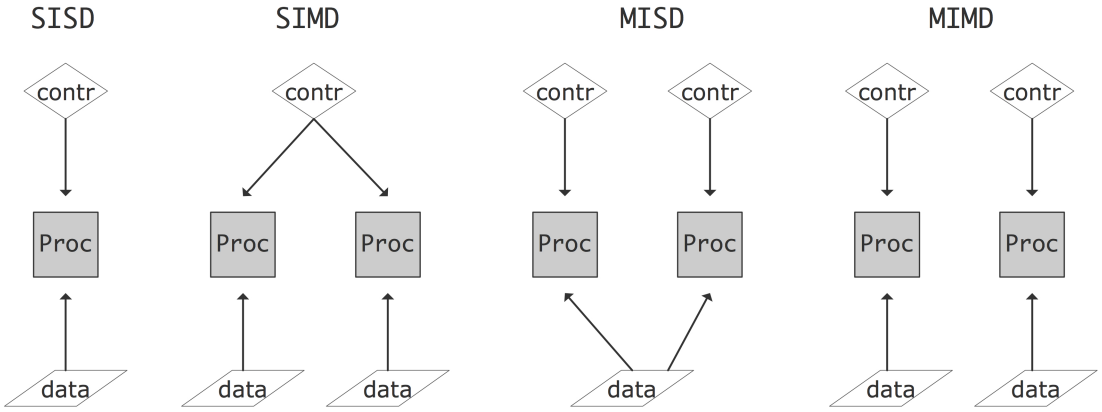


Figure 2.6: The four classes of the Flynn's taxonomy.

SISD Single Instruction Single Data: this is the traditional CPU architecture: at any one time only a single instruction is executed, operating on a single data item.

SIMD Single Instruction Multiple Data: in this computer type there can be multiple processors, each operating on its own data item, but they are all executing the same instruction on that data item. Vector computers (section 2.3.1.1) are typically also characterized as SIMD.

MISD Multiple Instruction Single Data. No architectures answering to this description exist; one could argue that redundant computations for safety-critical applications are an example of MISD.

MIMD Multiple Instruction Multiple Data: here multiple CPUs operate on multiple data items, each executing independent instructions. Most current parallel computers are of this type.

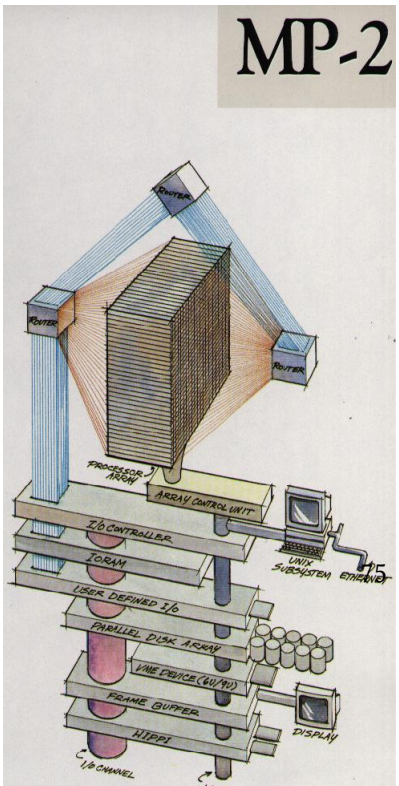
We will now discuss SIMD and MIMD architectures in more detail.

2.3.1 SIMD

Parallel computers of the SIMD type apply the same operation simultaneously to a number of data items. The design of the CPUs of such a computer can be quite simple, since the arithmetic unit does not need separate logic and instruction decoding units: all CPUs execute the same operation in lock step. This makes SIMD computers excel at operations on arrays, such as

```
for (i=0; i<N; i++) a[i] = b[i]+c[i];
```

and, for this reason, they are also often called *array processors*. Scientific codes can often be written so that a large fraction of the time is spent in array operations.



On the other hand, there are operations that can not can be executed efficiently on an array processor. For instance, evaluating a number of terms of a recurrence $x_{i+1} = ax_i + b_i$ involves that many additions and

multiplications, but they alternate, so only one operation of each type can be processed at any one time. There are no arrays of numbers here that are simultaneously the input of an addition or multiplication.

In order to allow for different instruction streams on different parts of the data, the processor would have a 'mask bit' that could be set to prevent execution of instructions. In code, this typically looks like

```
where (x>0) {
  x[i] = sqrt(x[i])
```

The programming model where identical operations are applied to a number of data items simultaneously, is known as *data parallelism*.

Such array operations can occur in the context of physics simulations, but another important source is graphics applications. For this application, the processors in an array processor can be much weaker than the processor in a PC: often they are in fact bit processors, capable of operating on only a single bit at a time. Along these lines, *ICL* had the 4096 processor DAP [114] in the 1980s, and *Goodyear* built a 16K processor MPP [9] in the 1970s.

Later, the *Connection Machine* (CM-1, CM-2, CM-5) were quite popular. While the first Connection Machine had bit processors (16 to a chip), the later models had traditional processors capable of floating point arithmetic, and were not true SIMD architectures. All were based on a hyper-cube interconnection network; see section 2.7.5. Another manufacturer that had a commercially successful array processor was *MasPar*; figure 2.7 illustrates the architecture. You clearly see the single control unit for a square array of processors, plus a network for doing global operations.

Supercomputers based on array processing do not exist anymore, but the notion of SIMD lives on in various guises. For instance, GPUs are SIMD-based, enforced through their *CUDA* programming language. Also, the *Intel Xeon Phi* has a strong SIMD component. While early SIMD architectures were motivated by minimizing the number of transistors necessary, these modern co-processors are motivated by *power efficiency* considerations. Processing instructions (known as *instruction issue*) is actually expensive compared to a floating point operation, in time, energy, and chip real estate needed. Using SIMD is then a way to economize on the last two measures.

2.3.1.1 Pipelining and pipeline processors

A number of computers have been based on a *vector processor* or *pipeline processor* design. The first commercially successful supercomputers, the Cray-1 and the Cyber-205 were of this type. In recent times, the Cray-X1 and the NEC SX series have featured vector pipes. The 'Earth Simulator' computer [171], which led the TOP500 (section 2.11.4) for 3 years, was based on NEC SX processors. The general idea behind pipelining was described in section 1.2.1.3.

While supercomputers based on pipeline processors are in a distinct minority, pipelining is now mainstream in the superscalar CPUs that are the basis for *clusters*. A typical CPU has pipelined floating point units, often with separate units for addition and multiplication; see section 1.2.1.3.

However, there are some important differences between pipelining in a modern superscalar CPU and in, more old-fashioned, vector units. The pipeline units in these vector computers are not integrated floating point units in the CPU, but can better be considered as attached vector units to a CPU that itself has a floating point unit. The vector unit has *vector registers*[2] with a typical length of 64 floating point numbers; there is typically no 'vector cache'. The logic in vector units is also simpler, often addressable by explicit vector instructions. Superscalar CPUs, on the other hand, are fully integrated in the CPU and geared towards exploiting data streams in unstructured code.

2.3.1.2 True SIMD in CPUs and GPUs

True SIMD array processing can be found in modern CPUs and GPUs, in both cases inspired by the parallelism that is needed in graphics applications.

Modern CPUs from Intel and AMD, as well as PowerPC chips, have *vector instructions* that can perform multiple instances of an operation simultaneously. On Intel processors this is known as *SIMD Streaming Extensions (SSE)* or *Advanced Vector Extensions (AVX)*. These extensions were originally intended for graphics processing, where often the same operation needs to be performed on a large number of pixels. Often, the data has to be a total of, say, 128 bits, and this can be divided into two 64-bit reals, four 32-bit reals, or a larger number of even smaller chunks such as 4 bits.

The AVX instructions are based on up to 512-bit wide SIMD, that is, eight double precision floating point numbers can be processed simultaneously. Just as single floating point operations operate on data in registers (section 1.3.4), vector operations use *vector registers*. The locations in a vector register are sometimes referred to as *SIMD lanes*.

The use of SIMD is mostly motivated by power considerations. Decoding instructions can be more power consuming than executing them, so SIMD parallelism is a way to save power.

Current compilers can generate SSE or AVX instructions automatically; sometimes it is also possible for the user to insert pragmas, for instance with the Intel compiler:

```
void func(float *restrict c, float *restrict a,
          float *restrict b, int n)
{
#pragma vector always
  for (int i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Use of these extensions often requires data to be aligned with cache line boundaries (section 1.3.5.8), so there are special allocate and free calls that return aligned memory.

Version 4 of *OpenMP* also has directives for indicating SIMD parallelism.

Array processing on a larger scale can be found in *GPU*s. A GPU contains a large number of simple processors, ordered in groups of 32, typically. Each processor group is limited to executing the same instruction. Thus, this is true example of SIMD processing. For further discussion, see section 2.9.3.

2. The Cyber205 was an exception, with direct-to-memory architecture.

2.3.2 MIMD / SPMD computers

By far the most common parallel computer architecture these days is called Multiple Instruction Multiple Data (MIMD): the processors execute multiple, possibly differing instructions, each on their own data. Saying that the instructions differ does not mean that the processors actually run different programs: most of these machines operate in *Single Program Multiple Data (SPMD)* mode, where the programmer starts up the same executable on the parallel processors. Since the different instances of the executable can take differing paths through conditional statements, or execute differing numbers of iterations of loops, they will in general not be completely in sync as they were on SIMD machines. If this lack of synchronization is due to processors working on different amounts of data, it is called *load unbalance*, and it is a major source of less than perfect *speedup*; see section 2.10.

There is a great variety in MIMD computers. Some of the aspects concern the way memory is organized, and the network that connects the processors. Apart from these hardware aspects, there are also differing ways of programming these machines. We will see all these aspects below. Many machines these days are called *clusters*. They can be built out of custom or commodity processors (if they consist of PCs, running Linux, and connected with *Ethernet*, they are referred to as *Beowulf clusters* [90]); since the processors are independent they are examples of the MIMD or SPMD model.

2.3.3 The commoditization of supercomputers

In the 1980s and 1990s supercomputers were radically different from personal computer and mini or supermini computers such as the DEC PDP and VAX series. The SIMD vector computers had one (*CDC Cyber205* or *Cray-1*), or at most a few (*ETA-10*, *Cray-2*, *Cray X/MP*, *Cray Y/MP*), extremely powerful processors, often a vector processor. Around the mid-1990s clusters with thousands of simpler (micro) processors started taking over from the machines with relative small numbers of vector pipes (see `http://www.top500.org/lists/1994/11`). At first these microprocessors (*IBM Power series*, *Intel i860*, *MIPS*, *DEC Alpha*) were still much more powerful than 'home computer' processors, but later this distinction also faded to an extent. Currently, many of the most powerful clusters are powered by essentially the same Intel Xeon and AMD Opteron chips that are available on the consumer market. Others use IBM Power Series or other 'server' chips. See section 2.11.4 for illustrations of this history since 1993.

2.4 Different types of memory access

In the introduction we defined a parallel computer as a setup where multiple processors work together on the same problem. In all but the simplest cases this means that these processors need access to a joint pool of data. In the previous chapter you saw how, even on a single processor, memory can have a hard time keeping up with processor demands. For parallel machines, where potentially several processors want to access the same memory location, this problem becomes even worse. We can characterize parallel machines by the approach they take to the problem of reconciling multiple accesses, by multiple processes, to a joint pool of data.

The main distinction here is between *distributed memory* and *shared memory*. With distributed memory, each processor has its own physical memory, and more importantly its own *address space*. Thus, if two

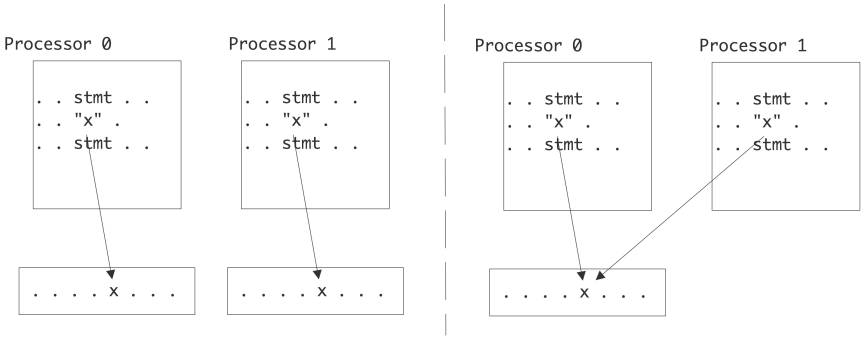


Figure 2.8: References to identically named variables in the distributed and shared memory case.

processors refer to a variable x, they access a variable in their own local memory. This is an instance of the SPMD model.

On the other hand, with shared memory, all processors access the same memory; we also say that they have a *shared address space*. See figure 2.8.

2.4.1 Symmetric Multi-Processors: Uniform Memory Access

Parallel programming is fairly simple if any processor can access any memory location. For this reason, there is a strong incentive for manufacturers to make architectures where processors see no difference between one memory location and another: any memory location is accessible to every processor, and the access times do not differ. This is called *Uniform Memory Access (UMA)*, and the programming model for architectures on this principle is often called *Symmetric Multi Processing (SMP)*.

There are a few ways to realize an SMP architecture. Current desktop computers can have a few processors accessing a shared memory through a single memory bus; for instance Apple markets a model with 2 six-core processors. Having a memory bus that is shared between processors works only for small numbers of processors; for larger numbers one can use a *crossbar* that connects multiple processors to multiple memory banks; see section 2.7.6.

On *multicore* processors there is uniform memory access of a different type: the cores typically have a *shared cache*, typically the L3 or L2 cache.

2.4.2 Non-Uniform Memory Access

The UMA approach based on shared memory is obviously limited to a small number of processors. The crossbar networks are expandable, so they would seem the best choice. However, in practice one puts processors with a local memory in a configuration with an exchange network. This leads to a situation where a processor can access its own memory fast, and other processors' memory slower. This is one case of so-called *NUMA*: a strategy that uses physically distributed memory, abandoning the uniform access time, but maintaining the logically shared address space: each processor can still access any memory location.

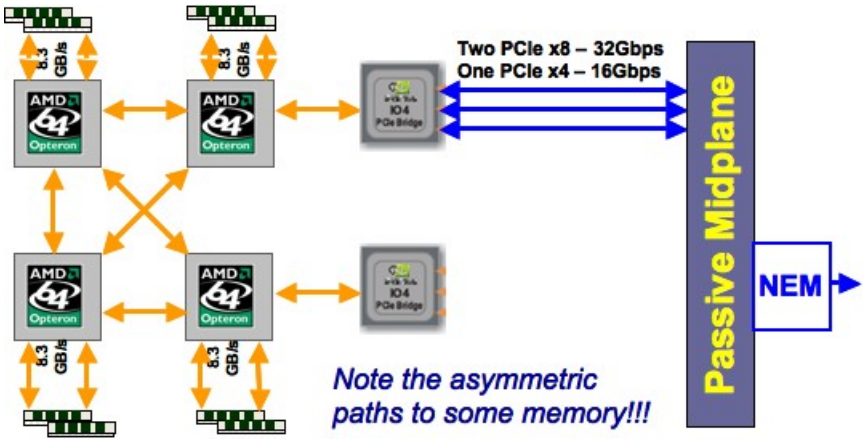


Figure 2.9: Non-uniform memory access in a four-socket motherboard.

2.4.2.1 *Affinity*

When we have NUMA, the question of where to place data, in relation to the process or thread that will access it, becomes important. This is known as *affinity*; if we look at it from the point of view of placing the processes or threads, it is called *process affinity*.

Figure 2.9 illustrates NUMA in the case of the four-socket motherboard of the *TACC Ranger cluster*. Each chip has its own memory (8Gb) but the motherboard acts as if the processors have access to a shared pool of 32Gb. Obviously, accessing the memory of another processor is slower than accessing local memory. In addition, note that each processor has three connections that could be used to access other memory, but the rightmost two chips use one connection to connect to the network. This means that accessing each other's memory can only happen through an intermediate processor, slowing down the transfer, and tying up that processor's connections.

2.4.2.2 *Coherence*

While the NUMA approach is convenient for the programmer, it offers some challenges for the system. Imagine that two different processors each have a copy of a memory location in their local (cache) memory. If one processor alters the content of this location, this change has to be propagated to the other processors. If both processors try to alter the content of the one memory location, the behavior of the program can become undetermined.

Keeping copies of a memory location synchronized is known as *cache coherence* (see section 1.4.1 for further details); a multi-processor system using it is sometimes called a 'cache-coherent NUMA' or *ccNUMA* architecture.

Taking NUMA to its extreme, it is possible to have a software layer that makes network-connected processors appear to operate on shared memory. This is known as *distributed shared memory* or *virtual shared memory*. In this approach a *hypervisor* offers a shared memory API, by translating system calls to distributed memory management. This shared memory API can be utilized by the *Linux kernel*, which can support 4096 threads.

Among current vendors only SGI (the *UV* line) and Cray (the *XE6*) market products with large scale NUMA. Both offer strong support for *Partitioned Global Address Space (PGAS)* languages; see section 2.6.5. There are vendors, such as *ScaleMP*, that offer a software solution to distributed shared memory on regular clusters.

2.4.3 Logically and physically distributed memory

The most extreme solution to the memory access problem is to offer memory that is not just physically, but that is also logically distributed: the processors have their own address space, and can not directly see another processor's memory. This approach is often called 'distributed memory', but this term is unfortunate, since we really have to consider the questions separately whether memory *is* distributed and whether is *appears* distributed. Note that NUMA also has physically distributed memory; the distributed nature of it is just not apparent to the programmer.

With logically *and* physically distributed memory, the only way one processor can exchange information with another is through passing information explicitly through the network. You will see more about this in section 2.6.3.3.

This type of architecture has the significant advantage that it can scale up to large numbers of processors: the *IBM BlueGene* has been built with over 200,000 processors. On the other hand, this is also the hardest kind of parallel system to program.

Various kinds of hybrids between the above types exist. In fact, most modern clusters will have NUMA nodes, but a distributed memory network between nodes.

2.5 Granularity of parallelism

In this section we look at parallelism from a point of how far to subdivide the work over processing elements. The concept we explore here is that of *granularity*: the balance between the amount of independent work per processing element, and how often processing elements need to synchronize. We talk of 'large grain parallelism' if there is a lot of work in between synchronization points, and 'small grain parallelism' if that amount of work is small. Obviously, in order for small grain parallelism to be profitable, the synchronization needs to be fast; with large grain parallelism we can tolerate most costly synchronization.

The discussion in this section will be mostly on a conceptual level; in section 2.6 we will go into some detail on actual parallel programming.

2.5.1 Data parallelism

It is fairly common for a program to have loops with a simple body that gets executed for all elements in a large data set:

```
for (i=0; i<1000000; i++)
  a[i] = 2*b[i];
```

Such code is considered an instance of *data parallelism* or *fine-grained parallelism*. If you had as many processors as array elements, this code would look very simple: each processor would execute the statement

```
a = 2*b
```

on its local data.

If your code consists predominantly of such loops over arrays, it can be executed efficiently with all processors in lockstep. Architectures based on this idea, where the processors can in fact *only* work in lockstep, have existed, see section 2.3.1. Such fully parallel operations on arrays appear in computer graphics, where every pixel of an image is processed independently. For this reason, GPUs (section 2.9.3) are strongly based on data parallelism.

The Compute-Unified Device Architecture (CUDA) language, invented by *NVidia*, allows for elegant expression of data parallelism. Later developed languages, such as *Sycl*, or libraries such as *Kokkos*, aim at similar expression, but are more geared towards heterogeneous parallelism.

Continuing the above example for a little bit, consider the operation

for $0 \leq i < \max$ **do**
$i_{\text{left}} = \mod(i-1, \max)$
$i_{\text{right}} = \mod(i+1, \max)$
$a_i = (b_{i_{\text{left}}} + b_{i_{\text{right}}})/2$

On a data parallel machine, that could be implemented as

`bleft` $\leftarrow$ **shiftright**(`b`)
`bright` $\leftarrow$ **shiftleft**(`b`)
`a` $\leftarrow$ (`bleft` + `bright`)$/2$

where the `shiftleft/right` instructions cause a data item to be sent to the processor with a number lower or higher by 1. For this second example to be efficient, it is necessary that each processor can communicate quickly with its immediate neighbors, and the first and last processor with each other.

In various contexts such a 'blur' operations in graphics, it makes sense to have operations on 2D data:

for $0 < i < m$ **do**
for $0 < j < n$ **do**
$a_{ij} \leftarrow (b_{ij-1} + b_{ij+1} + b_{i-1j} + b_{i+1j})$

and consequently processors have be able to move data to neighbors in a 2D grid.

2.5.2 Instruction-level parallelism

In *ILP*, the parallelism is still on the level of individual instructions, but these need not be similar. For instance, in

$a \leftarrow b + c$
$d \leftarrow e * f$

the two assignments are independent, and can therefore be executed simultaneously. This kind of parallelism is too cumbersome for humans to identify, but compilers are very good at this. In fact, identifying ILP is crucial for getting good performance out of modern *superscalar* CPUs.

2.5.3 Task-level parallelism

At the other extreme from data and instruction-level parallelism, *task parallelism* is about identifying whole subprograms that can be executed in parallel. As an example, a *search in a tree* data structure could be implemented as follows:

```
if optimal (root) then
    exit

else
    parallel: SearchInTree (leftchild),SearchInTree (rightchild)
                        Procedure SearchInTree(root)
```

The search tasks in this example are not synchronized, and the number of tasks is not fixed: it can grow arbitrarily. In practice, having too many tasks is not a good idea, since processors are most efficient if they work on just a single task. Tasks can then be scheduled as follows:

```
while there are tasks left do
    wait until a processor becomes inactive;
    spawn a new task on it
```

(There is a subtle distinction between the two previous pseudo-codes. In the first, tasks were self-scheduling: each task spawned off two new ones. The second code is an example of the *manager-worker paradigm*: there is one central task which lives for the duration of the code, and which spawns and assigns the worker tasks.)

Unlike in the data parallel example above, the assignment of data to processor is not determined in advance in such a scheme. Therefore, this mode of parallelism is most suited for thread-programming, for instance through the OpenMP library; section 2.6.2.

Let us consider a more serious example of task-level parallelism.

A finite element mesh is, in the simplest case, a collection of triangles that covers a 2D object. Since angles that are too acute should be avoided, the *Delauney mesh refinement* process can take certain triangles, and replace them by better shaped ones. This is illustrated in figure 2.10: the black triangles violate some angle condition, so either they themselves get subdivided, or they are joined with some neighboring ones (rendered in grey) and then jointly redivided.

In pseudo-code, this can be implemented as in figure 2.11. (This figure and code are to be found in [126], which also contains a more detailed discussion.)

It is clear that this algorithm is driven by a worklist (or *task queue*) data structure that has to be shared between all processes. Together with the dynamic assignment of data to processes, this implies that this type of *irregular parallelism* is suited to shared memory programming, and is much harder to do with distributed memory.

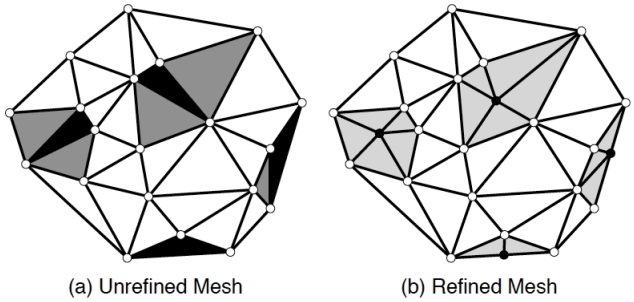


Figure 2.10: A mesh before and after refinement.

```
Mesh m = /* read in initial mesh */
WorkList wl;
wl.add(mesh.badTriangles());
while (wl.size() != 0) do
    Element e = wl.get(); //get bad triangle
    if (e no longer in mesh) continue;
    Cavity c = new Cavity(e);
    c.expand();
    c.retriangulate();
    mesh.update(c);
    wl.add(c.badTriangles());
```

Figure 2.11: Task queue implementation of Delauney refinement.

2.5.4 Conveniently parallel computing

In certain contexts, a simple, often single processor, calculation needs to be performed on many different inputs. Since the computations have no data dependencies and need not be done in any particular sequence, this is often called *embarrassingly parallel* or *conveniently parallel* computing. This sort of parallelism can happen at several levels. In examples such as calculation of the *Mandelbrot set* or evaluating moves in a *chess* game, a subroutine-level computation is invoked for many parameter values. On a coarser level it can be the case that a simple program needs to be run for many inputs. In this case, the overall calculation is referred to as a *parameter sweep*.

2.5.5 Medium-grain data parallelism

The above strict realization of data parallelism assumes that there are as many processors as data elements. In practice, processors will have much more memory than that, and the number of data elements is likely to be far larger than the processor count of even the largest computers. Therefore, arrays are grouped onto processors in subarrays. The code then looks like this:

```
my_lower_bound = // some processor-dependent number
my_upper_bound = // some processor-dependent number
```

```
for (i=my_lower_bound; i<my_upper_bound; i++)
  // the loop body goes here
```

This model has some characteristics of data parallelism, since the operation performed is identical on a large number of data items. It can also be viewed as task parallelism, since each processor executes a larger section of code, and does not necessarily operate on equal sized chunks of data.

2.5.6 Task granularity

In the previous subsections we considered different level of finding parallel work, or different ways of dividing up work so as to find parallelism. There is another way of looking at this: we define the *granularity* of a parallel scheme as the amount of work (or the task size) that a processing element can perform before having to communicate or synchronize with other processing elements.

In ILP we are dealing with very fine-grained parallelism, on the order of a single instruction or just a few instructions. In true task parallelism the granularity is much coarser.

The interesting case here is data parallelism, where we have the freedom to choose the task sizes. On SIMD machines we can choose a granularity of a single instruction, but, as you saw in section 2.5.5, operations can be grouped into medium-sized tasks. Thus, operations that are data parallel can be executed on distributed memory clusters, given the right balance between the number of processors and total problem size.

Exercise 2.20. Discuss choosing the right granularity for a data parallel operation such as averaging on a two-dimensional grid. Show that there is a *surface-to-volume* effect: the amount of communication is of a lower order than the computation. This means that, even if communication is much slower than computation, increasing the task size will still give a balanced execution.

Unfortunately, choosing a large task size to overcome slow communication may aggravate another problem: aggregating these operations may give tasks with varying running time, causing *load imbalance*. One solution here is to use an *overdecomposition* of the problem: create more tasks then there are processing elements, and assign multiple tasks to a processor (or assign tasks dynamically) to even out irregular running times. This is known as *dynamic scheduling*, and the examples in section 2.5.3 illustrate this; see also section 2.6.2.1. An example of *overdecomposition* in linear algebra is discussed in section 7.3.2.

2.6 Parallel programming

Parallel programming is more complicated than sequential programming. While for sequential programming most programming languages operate on similar principles (some exceptions such as functional or logic languages aside), there is a variety of ways of tackling parallelism. Let's explore some of the concepts and practical aspects.

There are various approaches to parallel programming. First of all, there does not seem to be any hope of a *parallelizing compiler* that can automagically transform a sequential program into a parallel one. Apart

from the problem of figuring out which operations are independent, the main problem is that the problem of locating data in a parallel context is very hard. A compiler would need to consider the whole code, rather than a subroutine at a time. Even then, results have been disappointing.

More productive is the approach where the user writes mostly a sequential program, but gives some indications about what computations can be parallelized, and how data should be distributed. Indicating parallelism of operations explicitly is done in OpenMP (section 2.6.2); indicating the data distribution and leaving parallelism to the compiler and runtime is the basis for PGAS languages (section 2.6.5). Such approaches work best with shared memory.

By far the hardest way to program in parallel, but with the best results in practice, is to expose the parallelism to the programmer and let the programmer manage everything explicitly. This approach is necessary in the case of distributed memory programming. We will have a general discussion of distributed programming in section 2.6.3.1; section 2.6.3.3 will discuss the MPI library.

2.6.1 Thread parallelism

As a preliminary to OpenMP (section 2.6.2), we will briefly go into 'threads'.

To explain what a *thread* is, we first need to get technical about what a *process* is. A unix process corresponds to the execution of a single program. Thus, it has in memory:

- The program code, in the form of machine language instructions;
- A *heap*, containing for instance arrays that were created with `malloc`;
- A stack with quick-changing information, such as the *program counter* that indicates what instruction is currently being executed, and data items with local scope, as well as intermediate results from computations.

This process can have multiple threads; these are similar in that they see the same program code and heap, but they have their own stack. Thus, a thread is an independent 'strand' of execution through a process.

Processes can belong to different users, or be different programs that a single user is running concurrently, so they have their own data space. On the other hand, threads are part of one process and therefore share the process heap. Threads can have some private data, for instance by have their own data stack, but their main characteristic is that they can collaborate on the same data.

2.6.1.1 The fork-join mechanism

Threads are dynamic, in the sense that they can be created during program execution. (This is different from the MPI model, where every processor run one process, and they are all created and destroyed at the same time.) When a program starts, there is one thread active: the *main thread*. Other threads are created by *thread spawning*, and the main thread can wait for their completion. This is known as the *fork-join* model; it is illustrated in figure 2.12. A group of threads that is forked from the same thread and active simultaneously is known as a *thread team*.

2.6.1.2 Hardware support for threads

Threads as they were described above are a software construct. Threading was possible before parallel computers existed; they were for instance used to handle independent activities in an OS. In the absence of parallel hardware, the OS would handle the threads through *multitasking* or *time slicing*: each

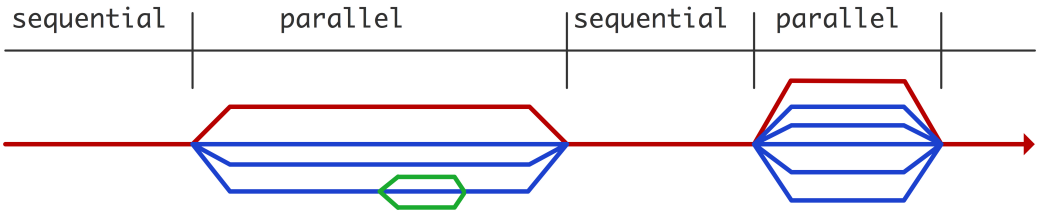


Figure 2.12: Thread creation and deletion during parallel execution.

thread would regularly get to use the CPU for a fraction of a second. (Technically, the Linux kernel treads processes and threads though the *task* concept; tasks are kept in a list, and are regularly activated or de-activated.)

This can lead to higher processor utilization, since the instructions of one thread can be processed while another thread is waiting for data. (On traditional CPUs, switching between threads is somewhat expensive (an exception is the *hyperthreading* mechanism) but on GPUs it is not, and in fact they *need* many threads to attain high performance.)

On modern *multicore* processors there is an obvious way of supporting threads: having one thread per core gives a parallel execution that uses your hardware efficiently. The shared memory allows the threads to all see the same data. This can also lead to problems; see section 2.6.1.5.

2.6.1.3 *Threads example*

The following example, which is strictly Unix-centric and will not work on Windows, is a clear illustration of the *fork-join* model. It uses the *pthreads* library to spawn a number of tasks that all update a global counter. Since threads share the same memory space, they indeed see and update the same memory location.

```
#include <stdlib.h>
#include <stdio.h>
#include "pthread.h"

int sum=0;

void adder() {
  sum = sum+1;
  return;
}

#define NTHREADS 50
int main() {
  int i;
  pthread_t threads[NTHREADS];
  printf("forking\n");
  for (i=0; i<NTHREADS; i++)
```

```
    if (pthread_create(threads+i,NULL,&adder,NULL)!=0) return i+1;
  printf("joining\n");
  for (i=0; i<NTHREADS; i++)
    if (pthread_join(threads[i],NULL)!=0) return NTHREADS+i+1;
  printf("Sum computed: %d\n",sum);

  return 0;
}
```

The fact that this code gives the right result is a coincidence: it only happens because updating the variable is so much quicker than creating the thread. (On a multicore processor the chance of errors will greatly increase.) If we artificially increase the time for the update, we will no longer get the right result:

```
void adder() {
  int t = sum; sleep(1); sum = t+1;
  return;
}
```

Now all threads read out the value of `sum`, wait a while (presumably calculating something) and then update.

This can be fixed by having a *lock* on the code region that should be 'mutually exclusive':

```
pthread_mutex_t lock;

void adder() {
  int t;
  pthread_mutex_lock(&lock);
  t = sum; sleep(1); sum = t+1;
  pthread_mutex_unlock(&lock);
  return;
}

int main() {
  ....
  pthread_mutex_init(&lock,NULL);
```

The lock and unlock commands guarantee that no two threads can interfere with each other's update.

2.6.1.4 Contexts

In the above example and its version with the `sleep` command we glanced over the fact that there were two types of data involved. First of all, the variable `s` was created outside the thread spawning part. Thus, this variable was *shared*.

On the other hand, the variable `t` was created once in each spawned thread. We call this *private* data.

The totality of all data that a thread can access is called its *context*. It contains private and shared data, as well as temporary results of computations that the thread is working on. (It also contains the program counter and stack pointer. If you don't know what those are, don't worry.)

It is quite possible to create more threads than a processor has cores, so a processor may need to switch between the execution of different threads. This is known as a *context switch*.

Context switches are not for free on regular CPUs, so they only pay off if the *granularity* of the threaded work is high enough. The exceptions to this story are:

- CPUs that have hardware support for multiple threads, for instance through *hyperthreading* (section 2.6.1.9), or as in the *Intel Xeon Phi* (section 2.9);
- GPUs, which in fact rely on fast context switching (section 2.9.3);
- certain other 'exotic' architectures such as the *Cray XMT* (section 2.8).

2.6.1.5 Race conditions, thread safety, and atomic operations

Shared memory makes life easy for the programmer, since every processor has access to all of the data: no explicit data traffic between the processor is needed. On the other hand, multiple processes/processors can also write to the same variable, which is a source of potential problems.

Suppose that two processes both try to increment an integer variable `I`:

Init: `I=0`
process 1: `I=I+2`
process 2: `I=I+3`

This is a legitimate activity if the variable is an accumulator for values computed by independent processes. The result of these two updates depends on the sequence in which the processors read and write the variable.

scenario 1.		scenario 2.		scenario 3.	
		I = 0			
read I = 0	read I = 0	read I = 0	read I = 0	read I = 0	
set I = 2	set I = 3	set I = 2	set I = 3	set I = 2	
write I = 2			write I = 3	write I = 2	
	write I = 3	write I = 2			read I = 2
					set I = 5
					write I = 5
I = 3		I = 2		I = 5	

Figure 2.13: Three executions of a data race scenario.

Figure 2.13 illustrates three scenarios. Such a scenario, where the final result depends on which thread executes first, is known as a *race condition* or *data race*. A formal definition would be:

> We talk of a a *data race* if there are two statements S_1, S_2,

- that are not causally related;
- that both access a location L; and
- at least one access is a write.

A very practical example of such conflicting updates is the inner product calculation:

```
for (i=0; i<1000; i++)
   sum = sum+a[i]*b[i];
```

Here the products are truly independent, so we could choose to have the loop iterations do them in parallel, for instance by their own threads. However, all threads need to update the same variable `sum`. This particular case of a data conflict is called *reduction*, and it is common enough that many threading systems have a dedicated mechanism for it.

Code that behaves the same whether it's executed sequentially or threaded is called *thread-safe*. As you can see from the above examples, a lack of thread safety is typically due to the treatment of shared data. This implies that the more your program uses local data, the higher the chance that it is thread safe. Unfortunately, sometimes the threads need to write to shared/global data.

There are essentially two ways of solving this problem. One is that we declare such updates of a shared variable a *critical section* of code. This means that the instructions in the critical section (in the inner product example 'read `sum` from memory, update it, write back to memory') can be executed by only one thread at a time. In particular, they need to be executed entirely by one thread before any other thread can start them so the ambiguity problem above will not arise. Of course, the above code fragment is so common that systems like OpenMP (section 2.6.2) have a dedicated mechanism for it, by declaring it a *reduction* operation.

Critical sections can for instance be implemented through the *semaphore* mechanism [48]. Surrounding each critical section there will be two atomic operations controlling a semaphore, a sign post. The first process to encounter the semaphore will lower it, and start executing the critical section. Other processes see the lowered semaphore, and wait. When the first process finishes the critical section, it executes the second instruction which raises the semaphore, allowing one of the waiting processes to enter the critical section.

The other way to resolve common access to shared data is to set a temporary *lock* on certain memory areas. This solution may be preferable, if common execution of the critical section is likely, for instance if it implements writing to a database or hash table. In this case, one process entering a critical section would prevent any other process from writing to the data, even if they might be writing to different locations; locking the specific data item being accessed is then a better solution.

The problem with locks is that they typically exist on the operating system level. This means that they are relatively slow. Since we hope that iterations of the inner product loop above would be executed at the speed of the floating point unit, or at least that of the memory bus, this is unacceptable.

One implementation of this is *transactional memory*, where the hardware itself supports atomic operations; the term derives from database transactions, which have a similar integrity problem. In transactional memory, a process will perform a normal memory update, unless the processor detects a conflict with an update from another process. In that case, the updates ('transactions') are canceled and retried

with one processor locking the memory and the other waiting for the lock. This is an elegant solution; however, canceling the transaction may carry a certain cost of *pipeline flushing* (section 1.2.5) and cache line invalidation (section 1.4.1).

2.6.1.6 Memory models and sequential consistency

The above signaled phenomenon of a *race condition* means that the result of some programs can be non-deterministic, depending on the sequence in which instructions are executed. There is a further factor that comes into play, and which is called the *memory model* that a processor and/or a language uses [2]. The memory model controls how the activity of one thread or core is seen by other threads or cores.

As an example, consider

initially: `A=B=0;`, then
process 1: `A=1; x = B;`
process 2: `B=1; y = A;`

As above, we have three scenarios, which we describe by giving a global sequence of statements:

scenario 1.	scenario 2.	scenario 3.
A ← 1	A ← 1	B ← 1
x ← B	B ← 1	y ← A
B ← 1	x ← B	A ← 1
y ← A	y ← A	x ← B
$x = 0, y = 1$	$x = 1, y = 1$	$x = 1, y = 0$

(In the second scenario, statements 1,2 can be reversed, as can 3,4, without change in outcome.)

The three different outcomes can be characterized as being computed by a global ordering on the statements that respects the local orderings. This is known as *sequential consistency*: the parallel outcome is consistent with a sequential execution that interleaves the parallel computations, respecting their local statement orderings.

Maintaining sequential consistency is expensive: it means that any change to a variable immediately needs to be visible on all other threads, or that any access to a variable on a thread needs to consult all other threads. We discussed this in section 1.4.1.

In a *relaxed memory model* it is possible to get a result that is not sequentially consistent. Suppose, in the above example, that the compiler decides to reorder the statements for the two processes, since the read and write are independent. In effect we get a fourth scenario:

scenario 4.
x ← B
y ← A
A ← 1
B ← 1
$x = 0, y = 0$

leading to the result $x = 0, y = 0$, which was not possible under the sequentially consistent model above. (There are algorithms for finding such dependencies [123].)

Sequential consistency implies that

```
integer n
n = 0
!$omp parallel shared(n)
n = n + 1
!$omp end parallel
```

should have the same effect as

```
n = 0
n = n+1 ! for processor 0
n = n+1 ! for processor 1
        ! et cetera
```

With sequential consistency it is no longer necessary to declare atomic operations or critical sections; however, this puts strong demands on the implementation of the model, so it may lead to inefficient code.

2.6.1.7 *Affinity*

Thread programming is very flexible, effectively creating parallelism as needed. However, a large part of this book is about the importance of data movement in scientific computations, and that aspect can not be ignored in thread programming.

In the context of a multicore processor, any thread can be scheduled to any core, and there is no immediate problem with this. However, if you care about high performance, this flexibility can have unexpected costs. There are various reasons why you want to certain threads to run only on certain cores. Since the OS is allowed to *migrate threads*, may be you simply want threads to stay in place.

- If a thread migrates to a different core, and that core has its own cache, you lose the contents of the original cache, and unnecessary memory transfers will occur.
- If a thread migrates, there is nothing to prevent the OS from putting two threads on one core, and leaving another core completely unused. This obviously leads to less than perfect speedup, even if the number of threads equals the number of cores.

We call *affinity* the mapping between threads (*thread affinity*) or processes (*process affinity*) and cores. Affinity is usually expressed as a *mask*: a description of the locations where a thread is allowed to run.

As an example, consider a two-socket node, where each socket has four cores.

With two threads and socket affinity we have the following affinity mask:

thread	socket 0	socket 1
0	0-1-2-3	
1		4-5-6-7

With core affinity the mask depends on the affinity type. The typical strategies are 'close' and 'spread'. With *close affinity*, the mask could be:

thread	socket 0	socket 1
0	0	
1	1	

Having two threads on the same socket means that they probably share an L2 cache, so this strategy is appropriate if they share data.

On the other hand, with *spread affinity* the threads are placed further apart:

thread	socket 0	socket 1
0	0	
1		4

This strategy is better for bandwidth-bound applications, since now each thread has the bandwidth of a socket, rather than having to share it in the 'close' case.

If you assign all cores, the close and spread strategies lead to different arrangements:

Close

socket 0	socket 1
0-1-2-3	
	4-5-6-7

Spread

socket 0	socket 1
0-2-4-6	
	1-3-5-7

Affinity and data access patterns Affinity can also be considered as a strategy of binding execution to data.

Consider this code:

```
for (i=0; i<ndata; i++) // this loop will be done by threads
  x[i] = ....
for (i=0; i<ndata; i++) // as will this one
  ... = .... x[i] ...
```

The first loop, by accessing elements of x, bring memory into cache or page table. The second loop accesses elements in the same order, so having a fixed affinity is the right decision for performance.

In other cases a fixed mapping is not the right solution:

```
for (i=0; i<ndata; i++) // produces loop
  x[i] = ....
for (i=0; i<ndata; i+=2) // use even indices
  ... = ... x[i] ...
for (i=1; i<ndata; i+=2) // use odd indices
  ... = ... x[i] ...
```

In this second example, either the program has to be transformed, or the programmer has to maintain in effect a *task queue*.

First touch It is natural to think of affinity in terms of 'put the execution where the data is'. However, in practice the opposite view sometimes makes sense. For instance, figure 2.9 showed how the shared memory of a cluster node can actually be distributed. Thus, a thread can be attached to a socket, but data can be allocated by the OS on any of the sockets. The mechanism that is often used by the OS is called the *first-touch* policy:

- When the program allocates data, the OS does not actually create it;
- instead, the memory area for the data is created the first time a thread accesses it;
- thus, the first thread to touch the area in effect causes the data to be allocated on the memory of its socket.

Exercise 2.21. Explain the problem with the following code:

```
// serial initialization
for (i=0; i<N; i++)
  a[i] = 0.;
#pragma omp parallel for
for (i=0; i<N; i++)
  a[i] = b[i] + c[i];
```

For an in-depth discussion of memory policies, see [130].

2.6.1.8 Cilk Plus

Other programming models based on threads exist. For instance, Intel *Cilk Plus* (http://www.cilkplus.org/) is a set of extensions of C/C++ with which a programmer can create threads.

Sequential code:

```
int fib(int n){
    if (n<2) return 1;
    else {
         int rst=0;
         rst += fib(n-1);
         rst += fib(n-2);
         return rst;
         }
}
```

Cilk code:

```
cilk int fib(int n){
         if (n<2) return 1;
         else {
              int rst=0;
              rst += cilk_spawn fib(n-1);
              rst += cilk_spawn fib(n-2);
              cilk_sync;
              return rst;
              }
}
```

Figure 2.14: Sequential and Cilk code for Fibonacci numbers.

Figure 2.14 shows the *Fibonacci numbers* calculation, sequentially and threaded with Cilk. In this example, the variable `rst` is updated by two, potentially independent threads. The semantics of this update, that is, the precise definition of how conflicts such as simultaneous writes are resolved, is defined by *sequential consistency*; see section 2.6.1.6.

2.6.1.9 Hyperthreading versus multi-threading

In the above examples you saw that the threads that are spawned during one program run essentially execute the same code, and have access to the same data. Thus, at a hardware level, a thread is uniquely determined by a small number of local variables, such as its location in the code (the *program counter*) and intermediate results of the current computation it is engaged in.

Hyperthreading is an Intel technology to let multiple threads use the processor truly simultaneously, so that part of the processor would be optimally used.

If a processor switches between executing one thread and another, it saves this local information of the one thread, and loads the information of the other. The cost of doing this is modest compared to running a whole program, but can be expensive compared to the cost of a single instruction. Thus, hyperthreading may not always give a performance improvement.

Certain architectures have support for *multi-threading*. This means that the hardware actually has explicit storage for the local information of multiple threads, and switching between the threads can be very fast. This is the case on GPUs (section 2.9.3), and on the *Intel Xeon Phi* architecture, where each core can support up to four threads.

However, the hyperthreads share the functional units of the core, that is, the arithmetic processing. Therefore, multiple active threads will not give a proportional increase in performance for computation-dominated codes. Most gain is expected in the case where the threads are heterogeneous in character.

2.6.2 OpenMP

OpenMP is an extension to the programming languages C and Fortran. Its main approach to parallelism is the parallel execution of loops: based on *compiler directives*, a preprocessor can schedule the parallel execution of the loop iterations.

Since OpenMP is based on *threads*, it features *dynamic parallelism*: the number of execution streams operating in parallel can vary from one part of the code to another. Parallelism is declared by creating parallel regions, for instance indicating that all iterations of a loop nest are independent, and the runtime system will then use whatever resources are available.

OpenMP is not a language, but an extension to the existing C and Fortran languages. It mostly operates by inserting directives into source code, which are interpreted by the compiler. It also has a modest number of library calls, but these are not the main point, unlike in MPI (section 2.6.3.3). Finally, there is a runtime system that manages the parallel execution.

OpenMP has an important advantage over MPI in its programmability: it is possible to start with a sequential code and transform it by *incremental parallelization*. By contrast, turning a sequential code into a distributed memory MPI program is an all-or-nothing affair.

Many compilers, such as *gcc* or the Intel compiler, support the OpenMP extensions. In Fortran, OpenMP directives are placed in comment statements; in C, they are placed in `#pragma` CPP directives, which indicate compiler specific extensions. As a result, OpenMP code still looks like legal C or Fortran to a compiler that does not support OpenMP. Programs need to be linked to an OpenMP runtime library, and their behavior can be controlled through environment variables.

For more information about OpenMP, see [32] and http://openmp.org/wp/.

2.6.2.1 *OpenMP examples*

The simplest example of OpenMP use is the parallel loop.

```
#pragma omp parallel for
for (i=0; i<ProblemSize; i++) {
  a[i] = b[i];
}
```

Clearly, all iterations can be executed independently and in any order. The pragma CPP directive then conveys this fact to the compiler.

Some loops are fully parallel conceptually, but not in implementation:

```
for (i=0; i<ProblemSize; i++) {
  t = b[i]*b[i];
  a[i] = sin(t) + cos(t);
}
```

Here it looks as if each iteration writes to, and reads from, a shared variable `t`. However, `t` is really a temporary variable, local to each iteration. Code that should be parallelizable, but is not due to such constructs, is called not *thread safe*.

OpenMP indicates that the temporary is private to each iteration as follows:

```
#pragma omp parallel for shared(a,b), private(t)
  for (i=0; i<ProblemSize; i++) {
    t = b[i]*b[i];
    a[i] = sin(t) + cos(t);
  }
```

If a scalar *is* indeed shared, OpenMP has various mechanisms for dealing with that. For instance, shared variables commonly occur in *reduction operations*:

```
  sum = 0;
#pragma omp parallel for reduction(+:sum)
  for (i=0; i<ProblemSize; i++) {
    sum = sum + a[i]*b[i];
  }
```

As you see, a sequential code can be parallelized with minimal effort.

The assignment of iterations to threads is done by the runtime system, but the user can guide this assignment. We are mostly concerned with the case where there are more iterations than threads: if there are P threads and N iterations and $N > P$, how is iteration i going to be assigned to a thread?

The simplest assignment uses *round-robin task scheduling*, a *static scheduling* strategy where thread p gets iterations $p \times (N/P), \ldots, (p+1) \times (N/P) - 1$. This has the advantage that if some data is reused between iterations, it will stay in the data cache of the processor executing that thread. On the other hand, if the iterations differ in the amount of work involved, the process may suffer from *load unbalance* with static scheduling. In that case, a *dynamic scheduling* strategy would work better, where each thread starts work on the next unprocessed iteration as soon as it finishes its current iteration. See the example in section 2.10.2.

You can control OpenMP scheduling of loop iterations with the `schedule` keyword; its values include `static` and `dynamic`. It is also possible to indicate a `chunksize`, which controls the size of the block of iterations that gets assigned together to a thread. If you omit the chunksize, OpenMP will divide the iterations into as many blocks as there are threads.

Exercise 2.22. Let's say there are t threads, and your code looks like

```
for (i=0; i<N; i++) {
  a[i] = // some calculation
}
```

If you specify a chunksize of 1, iterations $0, t, 2t, \ldots$ go to the first thread, $1, 1+t, 1+2t, \ldots$ to the second, et cetera. Discuss why this is a bad strategy from a performance point of view. Hint: look up the definition of *false sharing*. What would be a good chunksize?

2.6.3 Distributed memory programming through message passing

While OpenMP programs, and programs written using other shared memory paradigms, still look very much like sequential programs, this does not hold true for message passing code. Before we discuss the Message Passing Interface (MPI) library in some detail, we will take a look at this shift the way parallel code is written.

2.6.3.1 The global versus the local view in distributed programming

There can be a marked difference between how a parallel algorithm looks to an observer, and how it is actually programmed. Consider the case where we have an array of processors $\{P_i\}_{i=0..p-1}$, each containing one element of the arrays x and y, and P_i computes

$$\begin{cases} y_i \leftarrow y_i + x_{i-1} & i > 0 \\ y_i \text{ unchanged} & i = 0 \end{cases} \tag{2.4}$$

The global description of this could be

- Every processor P_i except the last sends its x element to P_{i+1};
- every processor P_i except the first receive an x element from their neighbor P_{i-1}, and
- they add it to their y element.

However, in general we can not code in these global terms. In the SPMD model (section 2.3.2) each processor executes the same code, and the overall algorithm is the result of these individual behaviors. The local program has access only to local data – everything else needs to be communicated with send and receive operations – and the processor knows its own number.

One possible way of writing this would be

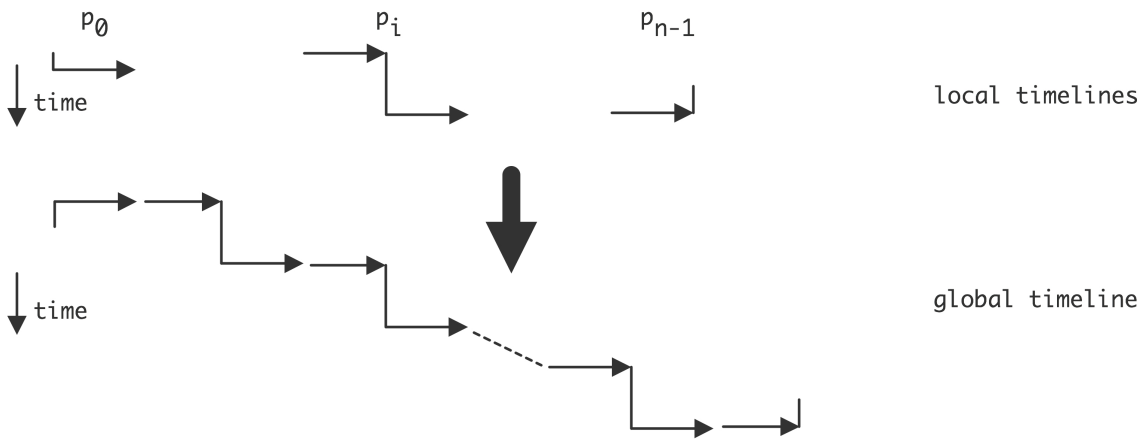


Figure 2.15: Local and resulting global view of an algorithm for sending data to the right.

- If I am processor 0 do nothing. Otherwise receive a y element from the left, add it to my x element.
- If I am the last processor do nothing. Otherwise send my y element to the right.

At first we look at the case where sends and receives are so-called *blocking communication* instructions: a send instruction does not finish until the sent item is actually received, and a receive instruction waits for the corresponding send. This means that sends and receives between processors have to be carefully paired. We will now see that this can lead to various problems on the way to an efficient code.

The above solution is illustrated in figure 2.15, where we show the local timelines depicting the local processor code, and the resulting global behavior. You see that the processors are not working at the same time: we get *serialized execution*.

What if we reverse the send and receive operations?

- If I am not the last processor, send my x element to the right;
- If I am not the first processor, receive an x element from the left and add it to my y element.

This is illustrated in figure 2.16 and you see that again we get a serialized execution, except that now the processors are activated right to left.

If the algorithm in equation 2.4 had been cyclic:

$$\begin{cases} y_i \leftarrow y_i + x_{i-1} & i = 1 \ldots n-1 \\ y_0 \leftarrow y_0 + x_{n-1} & i = 0 \end{cases} \tag{2.5}$$

the problem would be even worse. Now the last processor can not start its receive since it is blocked sending x_{n-1} to processor 0. This situation, where the program can not progress because every processor is waiting for another, is called *deadlock*.

The solution to getting an efficient code is to make as much of the communication happen simultaneously as possible. After all, there are no serial dependencies in the algorithm. Thus we program the algorithm as follows:

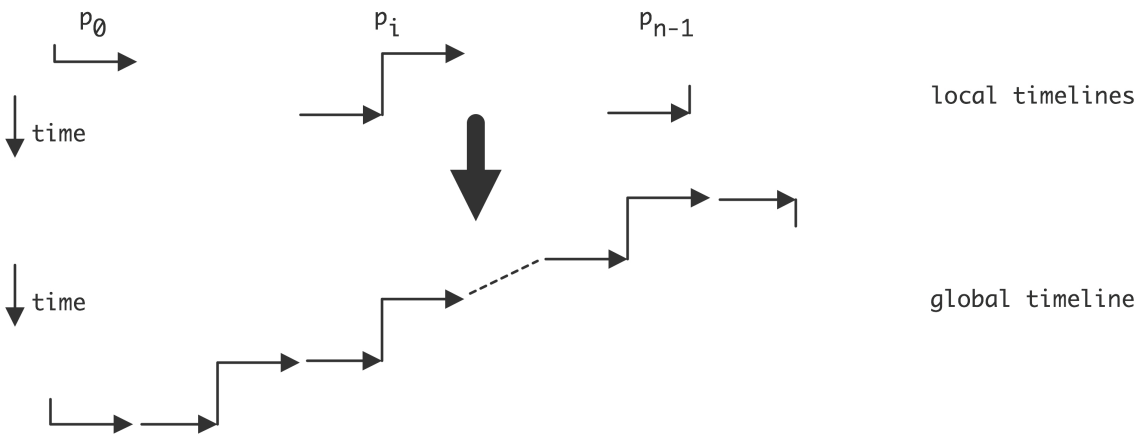


Figure 2.16: Local and resulting global view of an algorithm for sending data to the right.

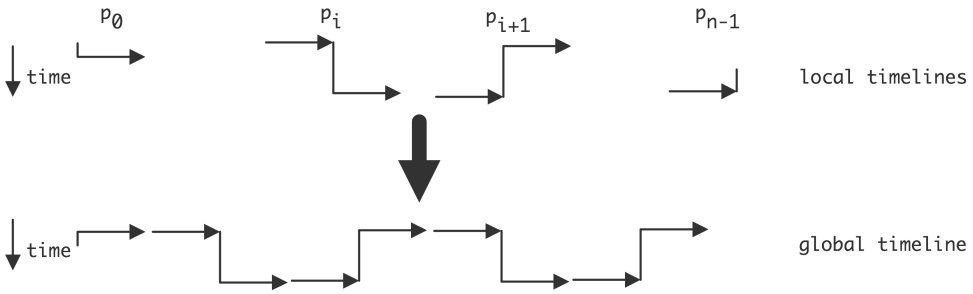


Figure 2.17: Local and resulting global view of an algorithm for sending data to the right.

- If I am an odd numbered processor, I send first, then receive;
- If I am an even numbered processor, I receive first, then send.

This is illustrated in figure 2.17, and we see that the execution is now parallel.

Exercise 2.23. Take another look at figure 2.3 of a parallel reduction. The basic actions are:

- receive data from a neighbor
- add it to your own data
- send the result on.

As you see in the diagram, there is at least one processor who does not send data on, and others may do a variable number of receives before they send their result on. Write node code so that an SPMD program realizes the distributed reduction. Hint: write each processor number in binary. The algorithm uses a number of steps that is equal to the length of this bitstring.

- Assuming that a processor receives a message, express the distance to the origin of that message in the step number.
- Every processor sends at most one message. Express the step where this happens in terms of the binary processor number.

2.6.3.2 Blocking and non-blocking communication

The reason for blocking instructions is to prevent accumulation of data in the network. If a send instruction were to complete before the corresponding receive started, the network would have to store the data somewhere in the mean time. Consider a simple example:

```
buffer = ... ;  // generate some data
send(buffer,0); // send to processor 0
buffer = ... ;  // generate more data
send(buffer,1); // send to processor 1
```

After the first send, we start overwriting the buffer. If the data in it hasn't been received, the first set of values would have to be buffered somewhere in the network, which is not realistic. By having the send operation block, the data stays in the sender's buffer until it is guaranteed to have been copied to the recipient's buffer.

One way out of the problem of sequentialization or deadlock that arises from blocking instruction is the use of *non-blocking communication* instructions, which include explicit buffers for the data. With non-blocking send instruction, the user needs to allocate a buffer for each send, and check when it is safe to overwrite the buffer.

```
buffer0 = ... ;   // data for processor 0
send(buffer0,0);  // send to processor 0
buffer1 = ... ;   // data for processor 1
send(buffer1,1);  // send to processor 1
...
// wait for completion of all send operations.
```

2.6.3.3 The MPI library

If OpenMP is the way to program shared memory, Message Passing Interface (MPI) [177] is the standard solution for programming distributed memory. MPI ('Message Passing Interface') is a specification for a library interface for moving data between processes that do not otherwise share data. The MPI routines can be divided roughly in the following categories:

- Process management. This includes querying the parallel environment and constructing subsets of processors.
- Point-to-point communication. This is a set of calls where two processes interact. These are mostly variants of the send and receive calls.
- Collective calls. In these routines, all processors (or the whole of a specified subset) are involved. Examples are the *broadcast* call, where one processor shares its data with every other processor, or the *gather* call, where one processor collects data from all participating processors.

Let us consider how the OpenMP examples can be coded in MPI. Note: this chapter is not a course in MPI programming, and consequently the examples will leave out many details of the MPI calls. If you want to learn MPI programming, consult for instance [86, 89, 87] or volume 2 of this series.

First of all, we no longer allocate

```
double a[ProblemSize];
```

but

```
double a[LocalProblemSize];
```

where the local size is roughly a $1/P$ fraction of the global size. (Practical considerations dictate whether you want this distribution to be as evenly as possible, or rather biased in some way.)

The parallel loop is trivially parallel, with the only difference that it now operates on a fraction of the arrays:

```
for (i=0; i<LocalProblemSize; i++) {
  a[i] = b[i];
}
```

However, if the loop involves a calculation based on the iteration number, we need to map that to the global value:

```
for (i=0; i<LocalProblemSize; i++) {
  a[i] = b[i]+f(i+MyFirstVariable);
}
```

(We will assume that each process has somehow calculated the values of `LocalProblemSize` and `MyFirstVariable`.) Local variables are now automatically local, because each process has its own instance:

```
for (i=0; i<LocalProblemSize; i++) {
  t = b[i]*b[i];
  a[i] = sin(t) + cos(t);
}
```

However, shared variables are harder to implement. Since each process has its own data, the local accumulation has to be explicitly assembled:

```
for (i=0; i<LocalProblemSize; i++) {
  s = s + a[i]*b[i];
}
MPI_Allreduce(s,globals,1,MPI_DOUBLE,MPI_SUM);
```

The 'reduce' operation sums together all local values `s` into a variable `globals` that receives an identical value on each processor. This is known as a *collective operation*.

Let us make the example slightly more complicated:

```
for (i=0; i<ProblemSize; i++) {
  if (i==0)
    a[i] = (b[i]+b[i+1])/2
  else if (i==ProblemSize-1)
    a[i] = (b[i]+b[i-1])/2
  else
    a[i] = (b[i]+b[i-1]+b[i+1])/3
}
```

If we had shared memory, we could write the following parallel code:

```
for (i=0; i<LocalProblemSize; i++) {
  bleft = b[i-1]; bright = b[i+1];
  a[i] = (b[i]+bleft+bright)/3
}
```

To turn this into valid distributed memory code, first we account for the fact that `bleft` and `bright` need to be obtained from a different processor for `i==0` (`bleft`), and for `i==LocalProblemSize-1` (`bright`). We do this with a exchange operation with our left and right neighbor processor:

```
// get bfromleft and bfromright from neighbor processors, then
for (i=0; i<LocalProblemSize; i++) {
  if (i==0) bleft=bfromleft;
    else bleft = b[i-1]
  if (i==LocalProblemSize-1) bright=bfromright;
    else bright = b[i+1];
  a[i] = (b[i]+bleft+bright)/3
}
```

Obtaining the neighbor values is done as follows. First we need to ask our processor number, so that we can start a communication with the processor with a number one higher and lower.

```
MPI_Comm_rank(MPI_COMM_WORLD,&myTaskID);
MPI_Sendrecv
   (/* to be sent:  */ &b[LocalProblemSize-1],
    /* destination  */ myTaskID+1,
    /* to be recvd: */ &bfromleft,
    /* source:      */ myTaskID-1,
    /* some parameters omitted */
   );
MPI_Sendrecv(&b[0],myTaskID-1,
    &bfromright, /* ... */ );
```

There are still two problems with this code. First, the sendrecv operations need exceptions for the first and last processors. This can be done elegantly as follows:

```
MPI_Comm_rank(MPI_COMM_WORLD,&myTaskID);
MPI_Comm_size(MPI_COMM_WORLD,&nTasks);
if (myTaskID==0) leftproc = MPI_PROC_NULL;
  else leftproc = myTaskID-1;
if (myTaskID==nTasks-1) rightproc = MPI_PROC_NULL;
  else rightproc = myTaskID+1;
MPI_Sendrecv( &b[LocalProblemSize-1], &bfromleft,  rightproc );
MPI_Sendrecv( &b[0],                  &bfromright, leftproc);
```

Exercise 2.24. There is still a problem left with this code: the boundary conditions from the original, global, version have not been taken into account. Give code that solves that problem.

MPI gets complicated if different processes need to take different actions, for example, if one needs to send data to another. The problem here is that each process executes the same executable, so it needs to contain both the send and the receive instruction, to be executed depending on what the rank of the process is.

```
if (myTaskID==0) {
  MPI_Send(myInfo,1,MPI_INT,/* to: */ 1,/* labeled: */,0,
    MPI_COMM_WORLD);
} else {
  MPI_Recv(myInfo,1,MPI_INT,/* from: */ 0,/* labeled: */,0,
    /* not explained here: */&status,MPI_COMM_WORLD);
}
```

2.6.3.4 *Blocking*

Although MPI is sometimes called the 'assembly language of parallel programming', for its perceived difficulty and level of explicitness, it is not all that hard to learn, as evinced by the large number of scientific codes that use it. The main issues that make MPI somewhat intricate to use are buffer management and blocking semantics.

These issues are related, and stem from the fact that, ideally, data should not be in two places at the same time. Let us briefly consider what happens if processor 1 sends data to processor 2. The safest strategy is for processor 1 to execute the send instruction, and then wait until processor 2 acknowledges that the data was successfully received. This means that processor 1 is temporarily blocked until processor 2 actually executes its receive instruction, and the data has made its way through the network. This is the standard behavior of the `MPI_Send` and `MPI_Recv` calls, which are said to use *blocking communication*.

Alternatively, processor 1 could put its data in a buffer, tell the system to make sure that it gets sent at some point, and later checks to see that the buffer is safe to reuse. This second strategy is called *non-blocking communication*, and it requires the use of a temporary buffer.

2.6.3.5 *Collective operations*

In the above examples, you saw the `MPI_Allreduce` call, which computed a global sum and left the result on each processor. There is also a local version `MPI_Reduce` which computes the result only on one processor. These calls are examples of *collective operations* or collectives. The collectives are:

reduction : each processor has a data item, and these items need to be combined arithmetically with an addition, multiplication, max, or min operation. The result can be left on one processor, or on all, in which case we call this an **allreduce** operation.

broadcast : one processor has a data item that all processors need to receive.

gather : each processor has a data item, and these items need to be collected in an array, without combining them in an operations such as an addition. The result can be left on one processor, or on all, in which case we call this an **allgather**.

scatter : one processor has an array of data items, and each processor receives one element of that array.

all-to-all : each processor has an array of items, to be scattered to all other processors.

Collective operations are blocking (see section 2.6.3.4), although MPI 3.0 (which is currently only a draft) will have non-blocking collectives. We will analyze the cost of collective operations in detail in section 7.1.

2.6.3.6 *Non-blocking communication*

In a simple computer program, each instruction takes some time to execute, in a way that depends on what goes on in the processor. In parallel programs the situation is more complicated. A send operation, in its

simplest form, declares that a certain buffer of data needs to be sent, and program execution will then stop until that buffer has been safely sent and received by another processor. This sort of operation is called a *non-local operation* since it depends on the actions of other processes, and a *blocking communication* operation since execution will halt until a certain event takes place.

Blocking operations have the disadvantage that they can lead to *deadlock*. In the context of message passing this describes the situation that a process is waiting for an event that never happens; for instance, it can be waiting to receive a message and the sender of that message is waiting for something else. Deadlock occurs if two processes are waiting for each other, or more generally, if you have a cycle of processes where each is waiting for the next process in the cycle. Example:

```
if ( /* this is process 0 */ )
   // wait for message from 1
else if ( /* this is process 1 */ )
   // wait for message from 0
```

A block receive here leads to deadlock. Even without deadlock, they can lead to considerable *idle time* in the processors, as they wait without performing any useful work. On the other hand, they have the advantage that it is clear when the buffer can be reused: after the operation completes, there is a guarantee that the data has been safely received at the other end.

The blocking behavior can be avoided, at the cost of complicating the buffer semantics, by using *non-blocking communication* operations. A non-blocking send (`MPI_Isend`) declares that a data buffer needs to be sent, but then does not wait for the completion of the corresponding receive. There is a second operation `MPI_Wait` that will actually block until the receive has been completed. The advantage of this decoupling of sending and blocking is that it now becomes possible to write:

```
MPI_ISend(somebuffer,&handle); // start sending, and
    // get a handle to this particular communication
{ ... }  // do useful work on local data
MPI_Wait(handle); // block until the communication is completed;
{ ... }  // do useful work on incoming data
```

With a little luck, the local operations take more time than the communication, and you have completely eliminated the communication time.

In addition to non-blocking sends, there are non-blocking receives. A typical piece of code then looks like

```
MPI_ISend(sendbuffer,&sendhandle);
MPI_IReceive(recvbuffer,&recvhandle);
{ ... }  // do useful work on local data
MPI_Wait(sendhandle); Wait(recvhandle);
{ ... }  // do useful work on incoming data
```

Exercise 2.25. Take another look at equation (2.5) and give pseudocode that solves the problem using non-blocking sends and receives. What is the disadvantage of this code over a blocking solution?

2.6.3.7 *MPI version 1 and 2 and 3*

The first MPI standard [157] had a number of notable omissions, which are included in the MPI 2 standard [88]. One of these concerned parallel input/output: there was no facility for multiple processes to

access the same file, even if the underlying hardware would allow that. A separate project MPI-I/O has now been rolled into the MPI-2 standard. We will discuss parallel I/O in this book.

A second facility missing in MPI, though it was present in *PVM* [50, 71] which predates MPI, is process management: there is no way to create new processes and have them be part of the parallel run.

Finally, MPI-2 has support for one-sided communication: one process puts data into the memory of another, without the receiving process doing an actual receive instruction. We will have a short discussion in section 2.6.3.8 below.

With MPI-3 the standard has gained a number of new features, such as non-blocking collectives, neighborhood collectives, and a profiling interface. The one-sided mechanisms have also been updated.

2.6.3.8 One-sided communication

The MPI way of writing matching send and receive instructions is not ideal for a number of reasons. First of all, it requires the programmer to give the same data description twice, once in the send and once in the receive call. Secondly, it requires a rather precise orchestration of communication if deadlock is to be avoided; the alternative of using asynchronous calls is tedious to program, requiring the program to manage a lot of buffers. Lastly, it requires a receiving processor to know how many incoming messages to expect, which can be tricky in irregular applications. Life would be so much easier if it was possible to pull data from another processor, or conversely to put it on another processor, without that other processor being explicitly involved.

This style of programming is further encouraged by the existence of *Remote Direct Memory Access (RDMA)* support on some hardware. An early example was the *Cray T3E*. These days, one-sided communication is widely available through its incorporation in the MPI-2 library; section 2.6.3.7.

Let us take a brief look at one-sided communication in MPI-2, using averaging of array values as an example:

$$\forall_i : a_i \leftarrow (a_i + a_{i-1} + a_{i+1})/3.$$

The MPI parallel code will look like

```
// do data transfer
a_local = (a_local+left+right)/3
```

It is clear what the transfer has to accomplish: the `a_local` variable needs to become the `left` variable on the processor with the next higher rank, and the `right` variable on the one with the next lower rank.

First of all, processors need to declare explicitly what memory area is available for one-sided transfer, the so-called 'window'. In this example, that consists of the `a_local`, `left`, and `right` variables on the processors:

```
MPI_Win_create(&a_local,...,&data_window);
MPI_Win_create(&left,....,&left_window);
MPI_Win_create(&right,....,&right_window);
```

The code now has two options: it is possible to push data out

```
target = my_tid-1;
MPI_Put(&a_local,...,target,right_window);
target = my_tid+1;
MPI_Put(&a_local,...,target,left_window);
```

or to pull it in

```
data_window = a_local;
source = my_tid-1;
MPI_Get(&right,...,data_window);
source = my_tid+1;
MPI_Get(&left,...,data_window);
```

The above code will have the right semantics if the Put and Get calls are blocking; see section 2.6.3.4. However, part of the attraction of one-sided communication is that it makes it easier to express communication, and for this, a non-blocking semantics is assumed.

The problem with non-blocking one-sided calls is that it becomes necessary to ensure explicitly that communication is successfully completed. For instance, if one processor does a one-sided *put* operation on another, the other processor has no way of checking that the data has arrived, or indeed that transfer has begun at all. Therefore it is necessary to insert a global barrier in the program, for which every package has its own implementation. In MPI-2 the relevant call is the `MPI_Win_fence` routine. These barriers in effect divide the program execution in *supersteps*; see section 2.6.8.

Another form of one-sided communication is used in the Charm++ package; see section 2.6.7.

2.6.4 Hybrid shared/distributed memory computing

Modern architectures are often a mix of shared and distributed memory. For instance, a cluster will be distributed on the level of the nodes, but sockets and cores on a node will have shared memory. One level up, each socket can have a shared L3 cache but separate L2 and L1 caches. Intuitively it seems clear that a mix of shared and distributed programming techniques would give code that is optimally matched to the architecture. In this section we will discuss such hybrid programming models, and discuss their efficacy.

A common setup of clusters uses distributed memory *nodes*, where each node contains several *sockets*, that share memory. This suggests using MPI to communicate between the nodes (*inter-node communication*) and OpenMP for parallelism on the node (*intra-node communication*).

In practice this is realized as follows:

- On each node a single MPI process is started (rather than one per core);
- This one MPI process then uses OpenMP (or another threading protocol) to spawn as many threads are there are independent sockets or cores on the node.
- The OpenMP threads can then access the shared memory of the node.

The alternative would be to have an MPI process on each core or socket and do all communication through message passing, even between processes that can see the same shared memory.

Remark *For reasons of* affinity *it may be desirable to start one MPI process per socket, rather than per node. This does not materially alter the above argument.*

This hybrid strategy may sound like a good idea but the truth is complicated.

- Message passing between MPI processes sounds like it's more expensive than communicating through shared memory. However, optimized versions of MPI can typically detect when processes are on the same node, and they will replace the message passing by a simple data copy. The only argument against using MPI is then that each process has its own data space, so there is memory overhead because each process has to allocate space for buffers and duplicates of the data that is copied.
- Threading is more flexible: if a certain part of the code needs more memory per process, an OpenMP approach could limit the number of threads on that part. On the other hand, flexible handling of threads incurs a certain amount of OS overhead that MPI does not have with its fixed processes.
- Shared memory programming is conceptually simple, but there can be unexpected performance pitfalls. For instance, the performance of two processes can now be impeded by the need for maintaining *cache coherence* and by *false sharing*.

On the other hand, the hybrid approach offers some advantage since it bundles messages. For instance, if two MPI processes on one node send messages to each of two processes on another node there would be four messages; in the hybrid model these would be bundled into one message.

Exercise 2.26. Analyze the discussion in the last item above. Assume that the bandwidth between the two nodes is only enough to sustain one message at a time. What is the cost savings of the hybrid model over the purely distributed model? Hint: consider bandwidth and latency separately.

This bundling of MPI processes may have an advantage for a deeper technical reason. In order to support a *handshake protocol*, each MPI process needs a small amount of buffer space for each other process. With a larger number of processes this can be a limitation, so bundling is attractive on high core count processors such as the *Intel Xeon Phi*.

The MPI library is explicit about what sort of threading it supports: you can query whether multi-threading is supported at all, whether all MPI calls have to originate from one thread or one thread at-a-time, or whether there is complete freedom in making MPI calls from threads.

2.6.5 Parallel languages

One approach to mitigating the difficulty of parallel programming is the design of languages that offer explicit support for parallelism. There are several approaches, and we will see some examples.

- Some languages reflect the fact that many operations in scientific computing are data parallel (section 2.5.1). Languages such as *High Performance Fortran (HPF)* (section 2.6.5.3) have an *array syntax*, where operations such as addition of arrays can be expressed as `A = B+C`. This syntax simplifies programming, but more importantly, it specifies operations at an abstract level, so that a lower level can make specific decision about how to handle parallelism. However, the data parallelism expressed in HPF is only of the simplest sort, where the data are contained in regular arrays. Irregular data parallelism is harder; the *Chapel* language (section 2.6.5.5) makes an attempt at addressing this.

- Another concept in parallel languages, not necessarily orthogonal to the previous, is that of Partitioned Global Address Space (PGAS) model: there is only one address space (unlike in the MPI model), but this address space is partitioned, and each partition has affinity with a thread or process. Thus, this model encompasses both SMP and distributed shared memory. A typical PGAS language, *Unified Parallel C (UPC)*, allows you to write programs that for the most part looks like regular C code. However, by indicating how the major arrays are distributed over processors, the program can be executed in parallel.

2.6.5.1 Discussion

Parallel languages hold the promise of making parallel programming easier, since they make communication operations appear as simple copies or arithmetic operations. However, by doing so they invite the user to write code that may not be efficient, for instance by inducing many small messages.

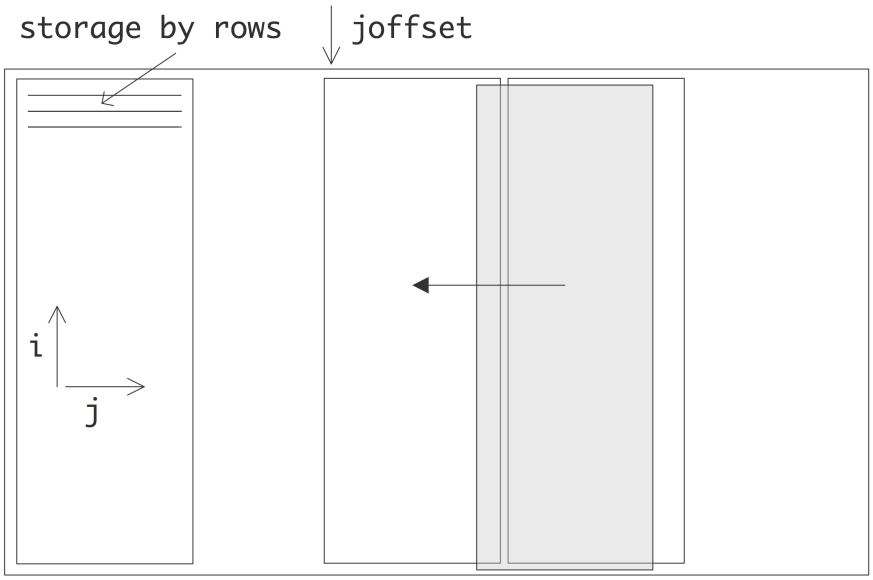


Figure 2.18: Data shift that requires communication.

As an example, consider arrays `a,b` that have been horizontally partitioned over the processors, and that are shifted (see figure 2.18):

```
for (i=0; i<N; i++)
  for (j=0; j<N/np; j++)
    a[i][j+joffset] = b[i][j+1+joffset]
```

If this code is executed on a shared memory machine, it will be efficient, but a naive translation in the distributed case will have a single number being communicated in each iteration of the `i` loop. Clearly, these can be combined in a single buffer send/receive operation, but compilers are usually unable to make this transformation. As a result, the user is forced to, in effect, re-implement the blocking that needs to be done in an MPI implementation:

```
for (i=0; i<N; i++)
  t[i] = b[i][N/np+joffset]
for (i=0; i<N; i++)
  for (j=0; j<N/np-1; j++) {
    a[i][j] = b[i][j+1]
    a[i][N/np] = t[i]
  }
```

On the other hand, certain machines support direct memory copies through global memory hardware. In that case, PGAS languages can be more efficient than explicit message passing, even with physically distributed memory.

2.6.5.2 Unified Parallel C

Unified Parallel C (UPC) [184] is an extension to the C language. Its main source of parallelism is *data parallelism*, where the compiler discovers independence of operations on arrays, and assigns them to separate processors. The language has an extended array declaration, which allows the user to specify whether the array is partitioned by blocks, or in a *round-robin* fashion.

The following program in UPC performs a vector-vector addition.

```
//vect_add.c
#include <upc_relaxed.h>
#define N 100*THREADS
shared int v1[N], v2[N], v1plusv2[N];
void main() {
  int i;
  for(i=MYTHREAD; i<N; i+=THREADS)
    v1plusv2[i]=v1[i]+v2[i];
}
```

The same program with an explicitly parallel loop construct:

```
//vect_add.c
#include <upc_relaxed.h>
#define N 100*THREADS
shared int v1[N], v2[N], v1plusv2[N];
void main()
{
  int i;
  upc_forall(i=0; i<N; i++; i)
    v1plusv2[i]=v1[i]+v2[i];
}
```

is comparable to UPC in spirit, but based on Java rather than on C.

2.6.5.3 High Performance Fortran

High Performance Fortran[3] (HPF) is an extension of Fortran 90 with constructs that support parallel computing, published by the High Performance Fortran Forum (HPFF). The HPFF was convened and chaired by Ken Kennedy of Rice University. The first version of the HPF Report was published in 1993.

Building on the array syntax introduced in Fortran 90, HPF uses a data parallel model of computation to support spreading the work of a single array computation over multiple processors. This allows efficient implementation on both SIMD and MIMD style architectures. HPF features included:

- New Fortran statements, such as FORALL, and the ability to create PURE (side effect free) procedures;
- The use of *compiler directives* for recommended distributions of array data;
- Extrinsic procedure interface for interfacing to non-HPF parallel procedures such as those using message passing;
- Additional library routines, including environmental inquiry, parallel prefix/suffix (e.g., 'scan'), data scattering, and sorting operations.

Fortran 95 incorporated several HPF capabilities. While some vendors did incorporate HPF into their compilers in the 1990s, some aspects proved difficult to implement and of questionable use. Since then, most vendors and users have moved to OpenMP-based parallel processing. However, HPF continues to have influence. For example the proposed BIT data type for the upcoming Fortran-2008 standard contains a number of new intrinsic functions taken directly from HPF.

2.6.5.4 Co-array Fortran

Co-array Fortran (CAF) is an extension to the Fortran 95/2003 language. The main mechanism to support parallelism is an extension to the array declaration syntax, where an extra dimension indicates the parallel distribution. For instance, in

```
Real,dimension(100),codimension[*] :: X
Real :: Y(100)[*]
Real :: Z(100,200)[10,0:9,*]
```

arrays `X,Y` have 100 elements on each processor. Array `Z` behaves as if the available processors are on a three-dimensional grid, with two sides specified and the third adjustable to accommodate the available processors.

Communication between processors is now done through copies along the (co-)dimensions that describe the processor grid. The Fortran 2008 standard includes co-arrays.

2.6.5.5 Chapel

Chapel [31] is a new parallel programming language[4] being developed by Cray Inc. as part of the DARPA-led High Productivity Computing Systems program (HPCS). Chapel is designed to improve the productivity of high-end computer users while also serving as a portable parallel programming model that can

3. This section quoted from Wikipedia
4. This section quoted from the Chapel homepage.

be used on commodity clusters or desktop multicore systems. Chapel strives to vastly improve the programmability of large-scale parallel computers while matching or beating the performance and portability of current programming models like MPI.

Chapel supports a multithreaded execution model via high-level abstractions for data parallelism, task parallelism, concurrency, and nested parallelism. Chapel's locale type enables users to specify and reason about the placement of data and tasks on a target architecture in order to tune for locality. Chapel supports global-view data aggregates with user-defined implementations, permitting operations on distributed data structures to be expressed in a natural manner. In contrast to many previous higher-level parallel languages, Chapel is designed around a multiresolution philosophy, permitting users to initially write very abstract code and then incrementally add more detail until they are as close to the machine as their needs require. Chapel supports code reuse and rapid prototyping via object-oriented design, type inference, and features for generic programming.

Chapel was designed from first principles rather than by extending an existing language. It is an imperative block-structured language, designed to be easy to learn for users of C, C++, Fortran, Java, Perl, Matlab, and other popular languages. While Chapel builds on concepts and syntax from many previous languages, its parallel features are most directly influenced by ZPL, High-Performance Fortran (HPF), and the Cray MTA's extensions to C and Fortran.

Here is vector-vector addition in Chapel:

```
const BlockDist= newBlock1D(bbox=[1..m], tasksPerLocale=...);
const ProblemSpace: domain(1, 64)) distributed BlockDist = [1..m];
var A, B, C: [ProblemSpace] real;
forall(a, b, c) in(A, B, C) do
  a = b + alpha * c;
```

2.6.5.6 Fortress

Fortress [65] is a programming language developed by Sun Microsystems. Fortress[5] aims to make parallelism more tractable in several ways. First, parallelism is the default. This is intended to push tool design, library design, and programmer skills in the direction of parallelism. Second, the language is designed to be more friendly to parallelism. Side-effects are discouraged because side-effects require synchronization to avoid bugs. Fortress provides transactions, so that programmers are not faced with the task of determining lock orders, or tuning their locking code so that there is enough for correctness, but not so much that performance is impeded. The Fortress looping constructions, together with the library, turns "iteration" inside out; instead of the loop specifying how the data is accessed, the data structures specify how the loop is run, and aggregate data structures are designed to break into large parts that can be effectively scheduled for parallel execution. Fortress also includes features from other languages intended to generally help productivity – test code and methods, tied to the code under test; contracts that can optionally be checked when the code is run; and properties, that might be too expensive to run, but can be fed to a theorem prover or model checker. In addition, Fortress includes safe-language features like checked array bounds, type checking, and garbage collection that have been proven-useful in Java. Fortress syntax is

5. This section quoted from the Fortress homepage.

designed to resemble mathematical syntax as much as possible, so that anyone solving a problem with math in its specification can write a program that is visibly related to its original specification.

2.6.5.7 X10

X10 is an experimental new language currently under development at IBM in collaboration with academic partners. The X10 effort is part of the IBM PERCS project (Productive Easy-to-use Reliable Computer Systems) in the DARPA program on High Productivity Computer Systems. The PERCS project is focused on a hardware-software co-design methodology to integrate advances in chip technology, architecture, operating systems, compilers, programming language and programming tools to deliver new adaptable, scalable systems that will provide an order-of-magnitude improvement in development productivity for parallel applications by 2010.

X10 aims to contribute to this productivity improvement by developing a new programming model, combined with a new set of tools integrated into Eclipse and new implementation techniques for delivering optimized scalable parallelism in a managed runtime environment. X10 is a type-safe, modern, parallel, distributed object-oriented language intended to be accessible to Java(TM) programmers. It is targeted to future low-end and high-end systems with nodes that are built out of multi-core SMP chips with non-uniform memory hierarchies, and interconnected in scalable cluster configurations. A member of the Partitioned Global Address Space (PGAS) family of languages, X10 highlights the explicit reification of locality in the form of places; lightweight activities embodied in async, future, foreach, and ateach constructs; constructs for termination detection (finish) and phased computation (clocks); the use of lock-free synchronization (atomic blocks); and the manipulation of global arrays and data structures.

2.6.5.8 Linda

As should be clear by now, the treatment of data is by far the most important aspect of parallel programming, far more important than algorithmic considerations. The programming system *Linda* [72, 73], also called a *coordination language*, is designed to address the data handling explicitly. Linda is not a language as such, but can, and has been, incorporated into other languages.

The basic concept of Linda is the *tuple space*: data is added to a pool of globally accessible information by adding a label to it. Processes then retrieve data by the label value, and without needing to know which processes added the data to the tuple space.

Linda is aimed primarily at a different computation model than is relevant for *HPC*: it addresses the needs of asynchronous communicating processes. However, is has been used for scientific computation [46]. For instance, in parallel simulations of the heat equation (section 4.3), processors can write their data into tuple space, and neighboring processes can retrieve their *halo region* without having to know its provenance. Thus, Linda becomes one way of implementing *one-sided communication*.

2.6.5.9 The Global Arrays library

The Global Arrays library (`http://www.emsl.pnl.gov/docs/global/`) is another example of *one-sided communication*, and in fact it predates MPI. This library has as its prime data structure *Cartesian product* arrays[6] , distributed over a processor grid of the same or lower dimension. Through library calls,

6. This means that if the array is three-dimensional, it can be described by three integers n_1, n_2, n_3, and each point has a coordinate (i_1, i_2, i_3) with $1 \leq i_1 \leq n_1$ et cetera.

any processor can access any sub-brick out of the array in either a `put` or `get` operation. These operations are non-collective. As with any one-sided protocol, a barrier sync is necessary to ensure completion of the sends/receives.

2.6.6 OS-based approaches

It is possible to design an architecture with a shared address space, and let the data movement be handled by the operating system. The Kendall Square computer [121] had an architecture name 'all-cache', where no data was directly associated with any processor. Instead, all data was considered to be cached on a processor, and moved through the network on demand, much like data is moved from main memory to cache in a regular CPU. This idea is analogous to the NUMA support in current SGI architectures.

2.6.7 Active messages

The MPI paradigm (section 2.6.3.3) is traditionally based on two-sided operations: each data transfer requires an explicit send and receive operation. This approach works well with relatively simple codes, but for complicated problems it becomes hard to orchestrate all the data movement. One of the ways to simplify consists of using *active messages*. This is used in the package *Charm++*[117].

With active messages, one processor can send data to another, without that second processor doing an explicit receive operation. Instead, the recipient declares code that handles the incoming data, a 'method' in objective orientation parlance, and the sending processor calls this method with the data that it wants to send. Since the sending processor in effect activates code on the other processor, this is also known as *remote method invocation*. A big advantage of this method is that overlap of communication and computation becomes easier to realize.

As an example, consider the matrix-vector multiplication with a tridiagonal matrix

$$\forall_i : y_i \leftarrow 2x_i - x_{i+1} - x_{i-1}.$$

See section 4.2.2 for an explanation of the origin of this problem in PDEs. Assuming that each processor has exactly one index i, the MPI code could look like:

```
if ( /* I am the first or last processor */ )
   n_neighbors = 1;
else
   n_neighbors = 2;

/* do the MPI_Isend operations on my local data */

sum = 2*local_x_data;
received = 0;
for (neighbor=0; neighbor<n_neighbors; neighbor++) {
   MPI_WaitAny( /* wait for any incoming data */ )
   sum = sum - /* the element just received */
   received++
   if (received==n_neighbors)
      local_y_data = sum
}
```

With active messages this looks like

```
void incorporate_neighbor_data(x) {
   sum = sum-x;
   if (received==n_neighbors)
      local_y_data = sum
}
sum = 2*local_xdata;
received = 0;
all_processors[myid+1].incorporate_neighbor_data(local_x_data);
all_processors[myid-1].incorporate_neighbor_data(local_x_data);
```

2.6.8 Bulk synchronous parallelism

The MPI library (section 2.6.3.3) can lead to very efficient code. The price for this is that the programmer needs to spell out the communication in great detail. On the other end of the spectrum, PGAS languages (section 2.6.5) ask very little of the programmer, but give not much performance in return. One attempt to find a middle ground is the *Bulk Synchronous Parallel (BSP)* model [185, 176]. Here the programmer needs to spell out the communications, but not their ordering.

The BSP model orders the program into a sequence of *supersteps*, each of which ends with a *barrier* synchronization. The communications that are started in one superstep are all asynchronous and rely on the barrier for their completion. This makes programming easier and removes the possibility of deadlock. Moreover, all communication are of the *one-sided communication* type.

Exercise 2.27. Consider the parallel summing example in section 2.1. Argue that a BSP implementation needs $\log_2 n$ supersteps.

Because of its synchronization of the processors through the barriers concluding the supersteps the BSP model can do a simple cost analysis of parallel algorithms.

Another aspect of the BSP model is its use of *overdecomposition* of the problem, where multiple processes are assigned to each processor, as well as *random placement* of data and tasks. This is motivated with a statistical argument that shows it can remedy *load imbalance*. If there are p processors and if in a superstep p remote accesses are made, with high likelihood some processor receives $\log p / \log\log p$ accesses, while others receive none. Thus, we have a load imbalance that worsens with increasing processor count. On the other hand, if $p \log p$ accesses are made, for instance because there are $\log p$ processes on each processor, the maximum number of accesses is $3 \log p$ with high probability. This means the load balance is within a constant factor of perfect.

The BSP model is implemented in BSPlib [105]. Other system can be said to be BSP-like in that they use the concept of supersteps; for instance Google's *Pregel* [145].

2.6.9 Data dependencies

If two statements refer to the same data item, we say that there is a *data dependency* between the statements. Such dependencies limit the extent to which the execution of the statements can be rearranged. The study of this topic probably started in the 1960s, when processors could execute statements *out of order* to increase throughput. The re-ordering of statements was limited by the fact that the execution

had to obey the *program order* semantics: the result had to be as if the statements were executed strictly in the order in which they appear in the program.

These issues of statement ordering, and therefore of data dependencies, arise in several ways:

- A *parallelizing compiler* has to analyze the source to determine what transformations are allowed;
- if you parallelize a sequential code with OpenMP directives, you have to perform such an analysis yourself.

Here are two types of activity that require such an analysis:

- When a loop is parallelized, the iterations are no longer executed in their program order, so we have to check for dependencies.
- The introduction of tasks also means that parts of a program can be executed in a different order from in which they appear in a sequential execution.

The easiest case of dependency analysis is that of detecting that loop iterations can be executed independently. Iterations are of course independent if a data item is read in two different iterations, but if the same item is read in one iteration and written in another, or written in two different iterations, we need to do further analysis.

Analysis of *data dependencies* can be performed by a compiler, but compilers take, of necessity, a conservative approach. This means that iterations may be independent, but can not be recognized as such by a compiler. Therefore, OpenMP shifts this responsibility to the programmer.

We will now discuss data depencies in some detail.

2.6.9.1 Types of data dependencies

The three types of dependencies are:

- flow dependencies, or 'read-after-write';
- anti dependencies, or 'write-after-read'; and
- output dependencies, or 'write-after-write'.

These dependencies can be studied in scalar code, and in fact compilers do this to determine whether statements can be rearranged, but we will mostly be concerned with their appearance in loops, since in scientific computation much of the work appears there.

Flow dependencies *Flow dependencies*, or read-afer-write, are not a problem if the read and write occur in the same loop iteration:

```
for (i=0; i<N; i++) {
  x[i] = .... ;
  .... = ... x[i] ... ;
}
```

On the other hand, if the read happens in a later iteration, there is no simple way to parallelize or *vectorize* the loop:

```
for (i=0; i<N; i++) {
  .... = ... x[i] ... ;
  x[i+1] = .... ;
}
```

This usually requires rewriting the code.

Exercise 2.28. Consider the code

```
for (i=0; i<N; i++) {
  a[i] = f(x[i]);
  x[i+1] = g(b[i]);
}
```

where `f()` and `g()` denote arithmetical expressions with out further dependencies on `x` or `i`. Show that this loop can be parallelized/vectorized if you are allowed to use a temporary array.

Anti dependencies The simplest case of an *anti dependency* or write-after-read is a reduction:

```
for (i=0; i<N; i++) {
  t = t + .....
}
```

This can be dealt with by explicit declaring the loop to be a reduction, or to use any of the other strategies in section 7.1.2.

If the read and write are on an array the situation is more complicated. The iterations in this fragment

```
for (i=0; i<N; i++) {
  x[i] = ... x[i+1] ... ;
}
```

can not be executed in arbitrary order as such. However, conceptually there is no dependency. We can solve this by introducing a temporary array:

```
for (i=0; i<N; i++)
  xtmp[i] = x[i];
for (i=0; i<N; i++) {
  x[i] = ... xtmp[i+1] ... ;
}
```

This is an example of a transformation that a compiler is unlikely to perform, since it can greatly affect the memory demands of the program. Thus, this is left to the programmer.

Output dependencies The case of an *output dependency* or write-after-write does not occur by itself: if a variable is written twice in sequence without an intervening read, the first write can be removed without changing the meaning of the program. Thus, this case reduces to a flow dependency.

Other output dependencies can also be removed. In the following code, `t` can be declared private, thereby removing the dependency.

```
for (i=0; i<N; i++) {
  t = f(i)
  s += t*t;
}
```

If the final value of `t` is wanted, the lastprivate can be used in OpenMP.

2.6.9.2 *Parallelizing nested loops*

In the above examples, data dependencies were non-trivial if in iteration i of a loop different indices appeared, such as i and $i+1$. Conversely, loops such as

```
for (int i=0; i<N; i++)
  x[i] = x[i]+f(i);
```

are simple to parallelize. Nested loops, however, take more thought. OpenMP has a 'collapse' directive for loops such as

```
for (int i=0; i<M; i++)
  for (int j=0; j<N; j++)
    x[i][j] = x[i][j] + y[i] + z[j];
```

Here, the whole i,j iteration space is parallel.

How is that with:

```
for (n = 0; n < NN; n++)
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      a[i] += B[i][j]*c[j] + d[n];
```

Exercise 2.29. Do a reuse analysis on this loop. Assume that `a,b,c` do not all fit in cache together.
Now assume that `c` and one row of `b` fit in cache, with a little room to spare. Can you find a loop interchange that will greatly benefit performance? Write a test to confirm this.

Analyzing this loop nest for parallelism, you see that the `j`-loop is a reduction, and the `n`-loop has flow dependencies: every `a[i]` is updated in every `n`-iteration. The conclusion is that you can only reasonable parallelize the `i`-loop.

Exercise 2.30. How does this parallelism analysis relate to the loop exchange from exercise 2.29? Is the loop after exchange still parallelizable?
If you speak OpenMP, confirm your answer by writing code that adds up the elements of `a`. You should get the same answer no matter the exchanges and the introduction of OpenMP parallelism.

2.6.10 Program design for parallelism

A long time ago it was thought that some magic combination of compiler and runtime system could transform an existing sequential program into a parallel one. That hope has long evaporated, so these days

a parallel program will have been written from the ground up as parallel. Of course there are different types of parallelism, and they each have their own implications for precisely how you design your parallel program. In this section we will briefly look into some of the issues.

2.6.10.1 Parallel data structures

One of the issues in parallel program design is the use of *Array-Of-Structures (AOS)* vs *Structure-Of-Arrays (SOA)*. In normal program design you often define a structure

```
struct { int number; double xcoord,ycoord; } _Node;
struct { double xtrans,ytrans} _Vector;
typedef struct _Node* Node;
typedef struct _Vector* Vector;
```

and if you need a number of them you create an array of such structures.

```
Node *nodes = (Node*) malloc( n_nodes*sizeof(struct _Node) );
```

This is the AOS design.

Now suppose that you want to parallelize an operation

```
void shift(Node the_point,Vector by) {
  the_point->xcoord += by->xtrans;
  the_point->ycoord += by->ytrans;
}
```

which is done in a loop

```
for (i=0; i<n_nodes; i++) {
  shift(nodes[i],shift_vector);
}
```

This code has the right structure for MPI programming (section 2.6.3.3), where every processor has its own local array of nodes. This loop is also readily parallelizable with OpenMP (section 2.6.2).

However, in the 1980s codes had to be substantially rewritten as it was realized that the AOS design was not good for vector computers. In that case you operands need to be contiguous, and so codes had to go to a SOA design:

```
node_numbers = (int*) malloc( n_nodes*sizeof(int) );
node_xcoords = // et cetera
node_ycoords = // et cetera
```

and you would iterate

```
for (i=0; i<n_nodes; i++) {
  node_xoords[i] += shift_vector->xtrans;
  node_yoords[i] += shift_vector->ytrans;
}
```

Oh, did I just say that the original SOA design was best for distributed memory programming? That meant that 10 years after the vector computer era everyone had to rewrite their codes again for clusters.

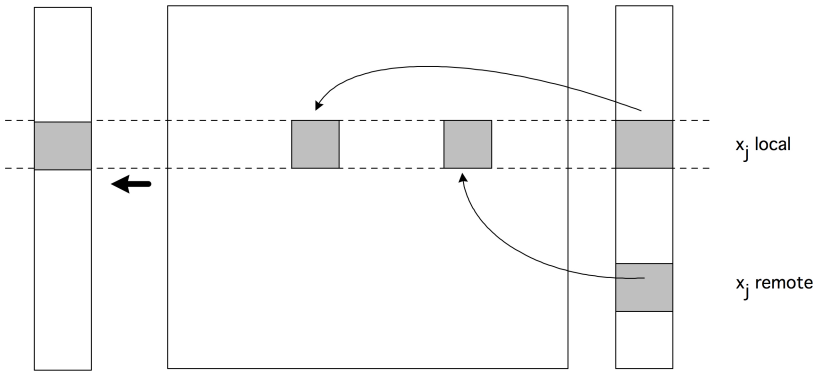


Figure 2.19: The parallel matrix-vector product with a blockrow distribution.

And of course nowadays, with increasing *SIMD width*, we need to go part way back to the AOS design. (There is some experimental software support for this transformation in the Intel *ispc* project, http://ispc.github.io/, which translates *SPMD* code to *SIMD*.)

2.6.10.2 Latency hiding

Communication between processors is typically slow, slower than data transfer from memory on a single processor, and much slower than operating on data. For this reason, it is good to think about the relative volumes of network traffic versus 'useful' operations when designing a parallel program. There has to be enough work per processor to offset the communication.

Another way of coping with the relative slowness of communication is to arrange the program so that the communication actually happens while some computation is going on. This is referred to as *overlapping computation with communication* or *latency hiding*.

For example, consider the parallel execution of a matrix-vector product $y = Ax$ (there will be further discussion of this operation in section 7.2.1). Assume that the vectors are distributed, so each processor p executes

$$\forall_{i\in I_p} : y_i = \sum_j a_{ij}x_j.$$

Since x is also distributed, we can write this as

$$\forall_{i\in I_p} : y_i = \left(\sum_{j \text{ local}} + \sum_{j \text{ not local}} \right) a_{ij}x_j.$$

This scheme is illustrated in figure 2.19. We can now proceed as follows:

- Start the transfer of non-local elements of x;
- Operate on the local elements of x while data transfer is going on;
- Make sure that the transfers are finished;
- Operate on the non-local elements of x.

Exercise 2.31. How much can you gain from overlapping computation and communication? Hint: consider the border cases where computation takes zero time and and there is only communication, and the reverse. Now consider the general case.

Of course, this scenario presupposes that there is software and hardware support for this overlap. MPI allows for this (see section 2.6.3.6), through so-called *asynchronous communication* or *non-blocking communication* routines. This does not immediately imply that overlap will actually happen, since hardware support is an entirely separate question.

2.7 Topologies

If a number of processors are working together on a single task, most likely they need to communicate data. For this reason there needs to be a way for data to make it from any processor to any other. In this section we will discuss some of the possible schemes to connect the processors in a parallel machine. Such a scheme is called a (processor) *topology*.

In order to get an appreciation for the fact that there is a genuine problem here, consider two simple schemes that do not 'scale up':

- *Ethernet* is a connection scheme where all machines on a network are on a single cable (see remark below). If one machine puts a signal on the wire to send a message, and another also wants to send a message, the latter will detect that the sole available communication channel is occupied, and it will wait some time before retrying its send operation. Receiving data on ethernet is simple: messages contain the address of the intended recipient, so a processor only has to check whether the signal on the wire is intended for it.
  The problems with this scheme should be clear. The capacity of the communication channel is finite, so as more processors are connected to it, the capacity available to each will go down. Because of the scheme for resolving conflicts, the average delay before a message can be started will also increase.
- In a *fully connected* configuration, each processor has one wire for the communications with each other processor. This scheme is perfect in the sense that messages can be sent in the minimum amount of time, and two messages will never interfere with each other. The amount of data that can be sent from one processor is no longer a decreasing function of the number of processors; it is in fact an increasing function, and if the network controller can handle it, a processor can even engage in multiple simultaneous communications.
  The problem with this scheme is of course that the design of the network interface of a processor is no longer fixed: as more processors are added to the parallel machine, the network interface gets more connecting wires. The network controller similarly becomes more complicated, and the cost of the machine increases faster than linearly in the number of processors.

Remark *The above description of Ethernet is of the original design. With the use of switches, especially in an HPC context, this description does not really apply anymore.*

It was initially thought that message collisions implied that ethernet would be inferior to other solutions such as IBM's token ring *network, which explicitly prevents collisions. It takes fairly sophisticated statistical analysis to prove that Ethernet works a lot better than was naively expected.*

In this section we will see a number of schemes that *can* be increased to large numbers of processors.

2.7.1 Some graph theory

The network that connects the processors in a parallel computer can conveniently be described with some elementary *graph theory* concepts. We describe the parallel machine with a graph where each processor is a node, and two nodes are connected if there is a direct connection between them. (We assume that connections are symmetric, so that the network is an *undirected graph*.)

We can then analyze two important concepts of this graph.

First of all, the *degree* of a node in a graph is the number of other nodes it is connected to. With the nodes representing processors, and the edges the wires, it is clear that a high degree is not just desirable for efficiency of computing, but also costly from an engineering point of view. We assume that all processors have the same degree.

Secondly, a message traveling from one processor to another, through one or more intermediate nodes, will most likely incur some delay at each stage of the path between the nodes. For this reason, the *diameter* of the graph is important. The diameter is defined as the maximum shortest distance, counting numbers of links, between any two nodes:

$$d(G) = \max_{i,j} |\text{shortest path between } i \text{ and } j|.$$

If d is the diameter, and if sending a message over one wire takes unit time, this means a message will always arrive in at most time d.

Exercise 2.32. Find a relation between the number of processors, their degree, and the diameter of the connectivity graph.

In addition to the question 'how long will a message from processor A to processor B take', we often worry about conflicts between two simultaneous messages: is there a possibility that two messages, under way at the same time, will need to use the same network link? In figure 2.20 we illustrate what happens if every processor p_i with $i < n/2$ send a message to $p_{i+n/2}$: there will be $n/2$ messages trying to get through the wire between $p_{n/2-1}$ and $p_{n/2}$. This sort of conflict is called *congestion* or *contention*. Clearly, the more

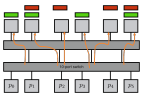


Figure 2.20: Contention for a network link due to simultaneous messages.

links a parallel computer has, the smaller the chance of congestion.

A precise way to describe the likelihood of congestion, is to look at the *bisection width*. This is defined as the minimum number of links that have to be removed to partition the processor graph into two unconnected graphs. For instance, consider processors connected as a linear array, that is, processor P_i is connected to P_{i-1} and P_{i+1}. In this case the bisection width is 1.

The bisection width w describes how many messages can, guaranteed, be under way simultaneously in a parallel computer. Proof: take w sending and w receiving processors. The w paths thus defined are disjoint: if they were not, we could separate the processors into two groups by removing only $w - 1$ links.

In practice, of course, more than w messages can be under way simultaneously. For instance, in a linear array, which has $w = 1$, $P/2$ messages can be sent and received simultaneously if all communication is between neighbors, and if a processor can only send or receive, but not both, at any one time. If processors can both send and receive simultaneously, P messages can be under way in the network.

Bisection width also describes *redundancy* in a network: if one or more connections are malfunctioning, can a message still find its way from sender to receiver?

While bisection width is a measure expressing a number of wires, in practice we care about the capacity through those wires. The relevant concept here is *bisection bandwidth*: the bandwidth across the bisection width, which is the product of the bisection width, and the capacity (in bits per second) of the wires. Bisection bandwidth can be considered as a measure for the bandwidth that can be attained if an arbitrary half of the processors communicates with the other half. Bisection bandwidth is a more realistic measure than the *aggregate bandwidth* which is sometimes quoted and which is defined as the total data rate if every processor is sending: the number of processors times the bandwidth of a connection times the number of simultaneous sends a processor can perform. This can be quite a high number, and it is typically not representative of the communication rate that is achieved in actual applications.

2.7.2 Busses

The first interconnect design we consider is to have all processors on the same *memory bus*. This design connects all processors directly to the same memory pool, so it offers a *UMA* or *SMP* model.

The main disadvantage of using a bus is the limited scalability, since only one processor at a time can do a memory access. To overcome this, we need to assume that processors are slower than memory, or that the processors have cache or other local memory to operate out of. In the latter case, maintaining *cache coherence* is easy with a bus by letting processors listen to all the memory traffic on the bus – a process known as *snooping*.

2.7.3 Linear arrays and rings

A simple way to hook up multiple processors is to connect them in a *linear array*: every processor has a number i, and processor P_i is connected to P_{i-1} and P_{i+1}. The first and last processor are possible exceptions: if they are connected to each other, we call the architecture a *ring network*.

This solution requires each processor to have two network connections, so the design is fairly simple.

Exercise 2.33. What is the bisection width of a linear array? Of a ring?

Exercise 2.34. With the limited connections of a linear array, you may have to be clever about how to program parallel algorithms. For instance, consider a 'broadcast' operation: processor 0 has a data item that needs to be sent to every other processor.
We make the following simplifying assumptions:

- a processor can send any number of messages simultaneously,

- but a wire can can carry only one message at a time; however,
- communication between any two processors takes unit time, regardless of the number of processors in between them.

In a fully connected network or a star network you can simply write
for $i = 1 \ldots N - 1$:
send the message to processor i
With the assumption that a processor can send multiple messages, this means that the operation is done in one step.
Now consider a linear array. Show that, even with this unlimited capacity for sending, the above algorithm runs into trouble because of congestion.
Find a better way to organize the send operations. Hint: pretend that your processors are connected as a binary tree. Assume that there are $N = 2^n - 1$ processors. Show that the broadcast can be done in $\log N$ stages, and that processors only need to be able to send a single message simultaneously.

This exercise is an example of *embedding* a 'logical' communication pattern in a physical one.

2.7.4 2D and 3D arrays

A popular design for parallel computers is to organize the processors in a two-dimensional or three-dimensional *Cartesian mesh*. This means that every processor has a coordinate (i, j) or (i, j, k), and it is connected to its neighbors in all coordinate directions. The processor design is still fairly simple: the number of network connections (the degree of the connectivity graph) is twice the number of space dimensions (2 or 3) of the network.

It is a fairly natural idea to have 2D or 3D networks, since the world around us is three-dimensional, and computers are often used to model real-life phenomena. If we accept for now that the physical model requires *nearest neighbor* type communications (which we will see is the case in section 4.2.3), then a mesh computer is a natural candidate for running physics simulations.

Exercise 2.35. What is the diameter of a 3D cube of $n \times n \times n$ processors? What is the bisection width? How does that change if you add wraparound torus connections?

Exercise 2.36. Your parallel computer has its processors organized in a 2D grid. The chip manufacturer comes out with a new chip with same clock speed that is dual core instead of single core, and that will fit in the existing sockets. Critique the following argument: 'the amount of work per second that can be done (that does not involve communication) doubles; since the network stays the same, the bisection bandwidth also stays the same, so I can reasonably expect my new machine to become twice as fast'.

Grid-based designs often have so-called *wrap-around* or *torus* connections, which connect the left and right sides of a 2D grid, as well as the top and bottom. This is illustrated in figure 2.21.

Some computer designs claim to be a grid of high dimensionality, for instance 5D, but not all dimensions are equal here. For instance, a 3D grid where each *node* is a quad-socket quad-core can be considered as a 5D grid. However, the last two dimensions are fully connected.

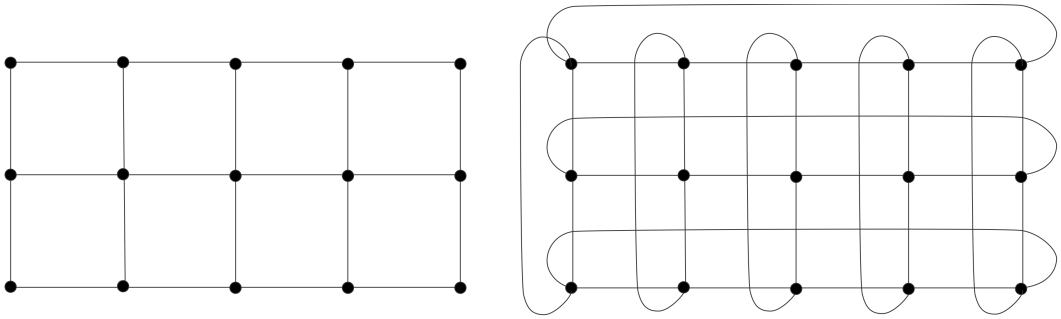


Figure 2.21: A 2D grid with torus connections.

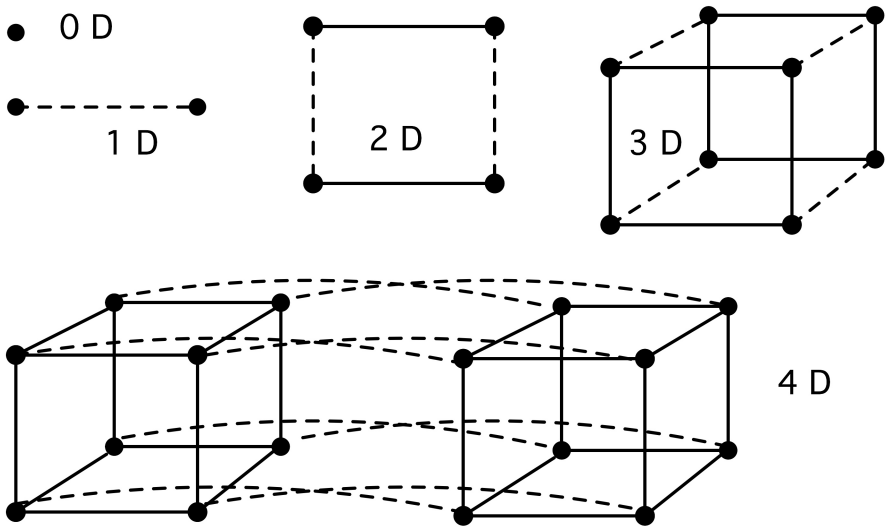


Figure 2.23: Hypercubes.

2.7.5 Hypercubes

Above we gave a hand-waving argument for the suitability of mesh-organized processors based on the prevalence of nearest neighbor communications. However, sometimes sends and receives between arbitrary processors occur. One example of this is the above-mentioned broadcast. For this reason, it is desirable to have a network with a smaller diameter than a mesh. On the other hand we want to avoid the complicated design of a fully connected network.

A good intermediate solution is the *hypercube* design. An n-dimensional hypercube computer has 2^n processors, with each processor connected to one other in each dimension; see figure 2.23.

An easy way to describe this is to give each processor an address consisting of d bits: we give each node of a hypercube a number that is the bit pattern describing

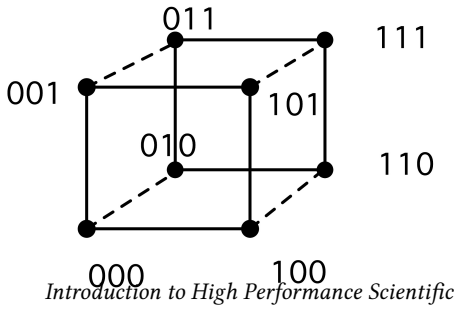


Figure 2.22: Numbering of the nodes of a hypercube.

its location in the cube; see figure 2.22.

With this numbering scheme, a processor is then connected to all others that have an address that differs by exactly one bit. This means that, unlike in a grid, a processor's neighbors do not have numbers that differ by 1 or $\sqrt{P}$, but by $1, 2, 4, 8, \ldots$.

The big advantages of a hypercube design are the small diameter and large capacity for traffic through the network.

Exercise 2.37. What is the diameter of a hypercube? What is the bisection width?

One disadvantage is the fact that the processor design is dependent on the total machine size. In practice, processors will be designed with a maximum number of possible connections, and someone buying a smaller machine then will be paying for unused capacity. Another disadvantage is the fact that extending a given machine can only be done by doubling it: other sizes than 2^p are not possible.

Exercise 2.38. Consider the parallel summing example of section 2.1, and give the execution time of a parallel implementation on a hypercube. Show that the theoretical speedup from the example is attained (up to a factor) for the implementation on a hypercube.

2.7.5.1 Embedding grids in a hypercube

Above we made the argument that mesh-connected processors are a logical choice for many applications that model physical phenomena. Hypercubes do not look like a mesh, but they have enough connections that they can simply pretend to be a mesh by ignoring certain connections.

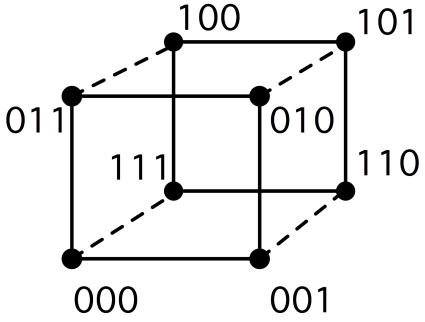
Let's say that we want the structure of a 1D array: we want processors with a numbering so that processor i can directly send data to $i-1$ and $i+1$. We can not use the obvious numbering of nodes as in figure 2.22. For instance, node 1 is directly connected to node 0, but has a distance of 2 to node 2. The right neighbor of node 3 in a ring, node 4, even has the maximum distance of 3 in this hypercube. Clearly we need to renumber the nodes in some way.

What we will show is that it's possible to walk through a hypercube, touching every corner exactly once, which is equivalent to embedding a 1D mesh in the hypercube.

1D Gray code:		0	1						
2D Gray code:	1D code and reflection:	0	1	⋮	1	0			
	append 0 and 1 bit:	0	0	⋮	1	1			
	2D code and reflection:	0	1	1	0	⋮	0	1	1	0
3D Gray code:		0	0	1	1	⋮	1	1	0	0
	append 0 and 1 bit:	0	0	0	0	⋮	1	1	1	1

Figure 2.24: Gray codes.

The basic concept here is a (binary reflected) *Gray code* [84]. This is a way of ordering the binary numbers $0 \ldots 2^d - 1$ as $g_0, \ldots g_{2^d-1}$ so that g_i and g_{i+1} differ in



only one bit. Clearly, the ordinary binary numbers do not satisfy this: the binary representations for 1 and 2 already differ in two bits. Why do Gray codes help us? Well, since g_i and g_{i+1} differ only in one bit, it means they are the numbers of nodes in the hypercube that are directly connected.

Figure 2.24 illustrates how to construct a Gray code. The procedure is recursive, and can be described informally as 'divide the cube into two subcubes, number the one subcube, cross over to the other subcube, and number its nodes in the reverse order of the first one'. The result for a 2D cube is in figure 2.25.

Since a Gray code offers us a way to embed a one-dimensional 'mesh' into a hypercube, we can now work our way up.

Exercise 2.39. Show how a square mesh of 2^{2d} nodes can be embedded in a hypercube by appending the bit patterns of the embeddings of two 2^d node cubes. How would you accommodate a mesh of $2^{d_1+d_2}$ nodes? A three-dimensional mesh of $2^{d_1+d_2+d_3}$ nodes?

2.7.6 Switched networks

Above, we briefly discussed fully connected processors. They are impractical if the connection is made by making a large number of wires between all the processors. There is another possibility, however, by connecting all the processors to a *switch* or switching network. Some popular network designs are the *crossbar*, the *butterfly exchange*, and the *fat tree*.

Switching networks are made out of switching elements, each of which have a small number (up to about a dozen) of inbound and outbound links. By hooking all processors up to some switching element, and having multiple stages of switching, it then becomes possible to connect any two processors by a path through the network.

2.7.6.1 Cross bar

The simplest switching network is a cross bar, an arrangement of n horizontal and vertical lines, with a switch element on each intersection that determines whether the lines are connected; see figure 2.26. If we designate the horizontal lines as inputs the vertical as outputs, this is clearly a way of having n inputs be mapped to n outputs. Every combination of inputs and outputs (sometimes called a 'permutation') is allowed.

One advantage of this type of network is that no connection blocks another. The main disadvantage is that the number of switching elements is n^2, a fast growing function of the number of processors n.

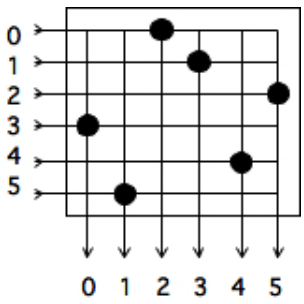


Figure 2.26: A simple cross bar connecting 6 inputs to 6 outputs.

2.7.6.2 *Butterfly exchange*

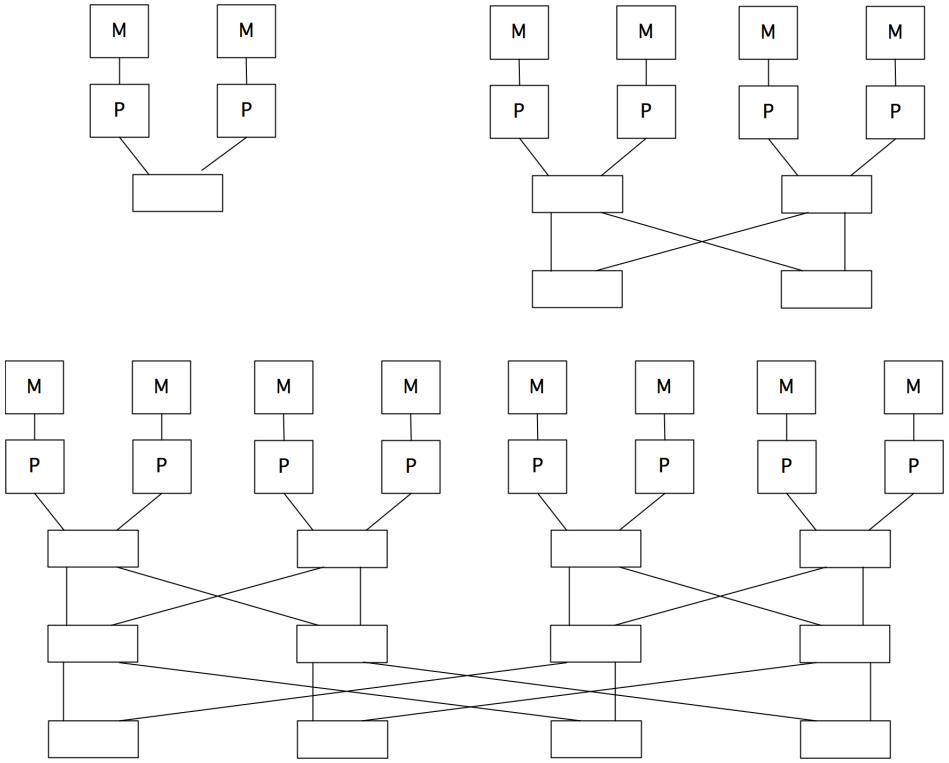


Figure 2.27: Butterfly exchange networks for 2,4,8 processors, each with a local memory.

Butterfly exchange networks are built out of small switching elements, and they have multiple stages: as the number of processors grows, the number of stages grows with it. Figure 2.27 shows shows butterfly networks to connect 2,4, and 8 processors, each with a local memory. (Alternatively, you could put all processors at one side of the network, and all memories at the other.)

As you can see in figure 2.28, butterfly exchanges allow several processors to access memory simultaneously. Also, their access times are identical, so exchange networks are a way of implementing a *UMA* architecture; see section 2.4.1. One computer that was based on a Butterfly exchange network was the *BBN Butterfly* (http://en.wikipedia.org/wiki/BBN_Butterfly). In section 2.7.7.1 we will see how these ideas are realized in a practical cluster.

Exercise 2.40. For both the simple cross bar and the butterfly exchange, the network needs to be expanded as the number of processors grows. Give the number of wires (of some unit length) and the number of switching elements that is needed in both cases to connect n processors and memories. What is the time that a data packet needs to go from memory to processor, expressed in the unit time that it takes to traverse a unit length of wire and the time to traverse a switching element?

Packet routing through a butterfly network is done based on considering the bits in the destination address.

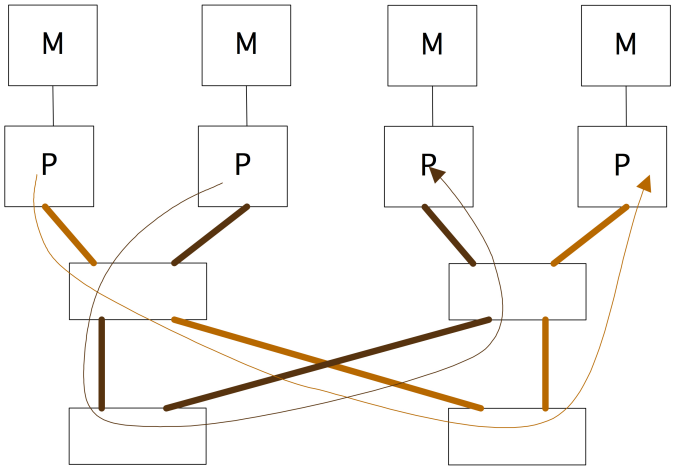


Figure 2.28: Two independent routes through a butterfly exchange network.

On the i-th level the i-th digit is considered; if this is 1, the left exit of the switch is taken, if 0, the right exit. This is illustrated in figure 2.29. If we attach the memories to the processors, as in figure 2.28, we need only two bits (to the last switch) but a further three bits to describe the reverse route.

2.7.6.3 *Fat-trees*

If we were to connect switching nodes like a tree, there would be a big problem with congestion close to the root since there are only two wires attached to the root note. Say we have a k-level tree, so there are 2^k leaf nodes. If all leaf nodes in the left subtree try to communicate with nodes in the right subtree, we have 2^{k-1} messages going through just one wire into the root, and similarly out through one wire. A fat-tree is a tree network where each level has the same total bandwidth, so that this congestion problem does not occur: the root will actually have 2^{k-1} incoming and outgoing wires attached [85]. Figure 2.30 shows this structure on the left; on the right is shown a cabinet of the Stampede cluster, with a *leaf switch* for top and bottom half of the cabinet.

The first successful computer architecture based on a fat-tree was the Connection Machines CM5.

In fat-trees, as in other switching networks, each message carries its own routing information. Since in a fat-tree the choices are limited to going up a level, or switching to the other subtree at the current level, a message needs to carry only as many bits routing information as there are levels, which is $\log_2 n$ for n processors.

Exercise 2.41. Show that the *bisection width of a fat tree* is $P/2$ where P is the number of processor leaf nodes. Hint: show that there is only one way of splitting a fat tree-connected set of processors into two connected subsets.

The theoretical exposition of fat-trees in [137] shows that fat-trees are optimal in some sense: it can deliver messages as fast (up to logarithmic factors) as any other network that takes the same amount of space to build. The underlying assumption of this statement is that switches closer to the root have to connect more wires, therefore take more components, and correspondingly are larger. This argument, while theoretically interesting, is of no practical significance, as the physical size of the network hardly

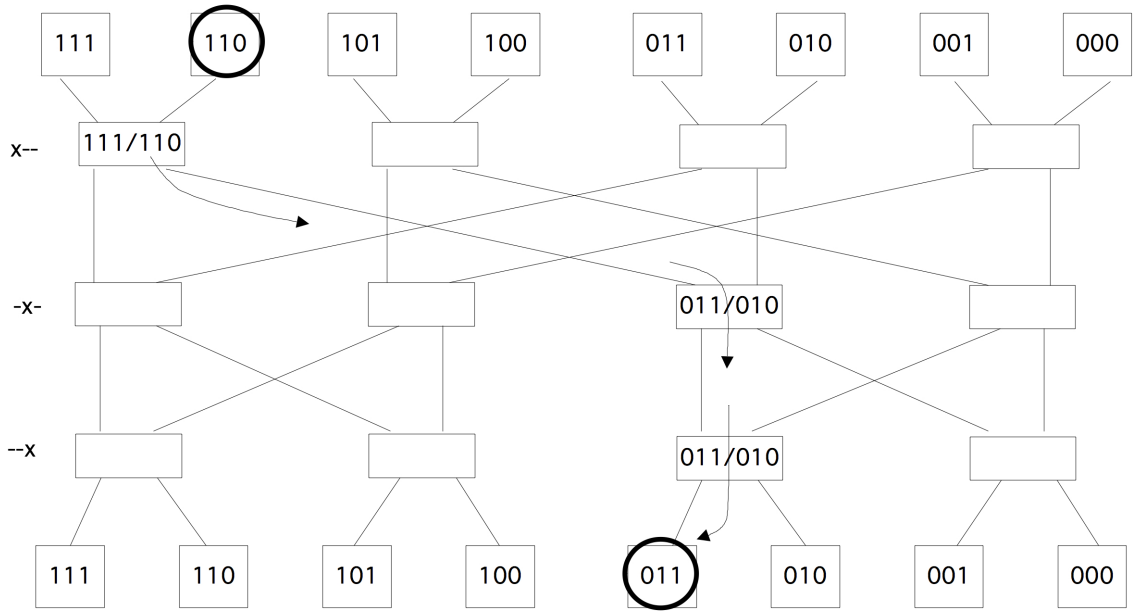


Figure 2.29: Routing through a three-stage butterfly exchange network.

plays a role in the biggest currently available computers that use fat-tree interconnect. For instance, in the *TACC Frontera cluster* at The University of Texas at Austin, there are only 6 *core switches* (that is, cabinents housing the top levels of the fat tree), connecting 91 processor cabinets.

A fat tree, as sketched above, would be costly to build, since for every next level a new, bigger, switch would have to be designed. In practice, therefore, a network with the characteristics of a fat-tree is constructed from simple switching elements; see figure 2.31. This network is equivalent in its bandwidth and routing possibilities to a fat-tree. Routing algorithms will be slightly more complicated: in a fat-tree, a data packet can go up in only one way, but here a packet has to know to which of the two higher switches to route.

This type of switching network is one case of a *Clos network* [35].

2.7.6.4 Over-subscription and contention

In practice, fat-tree networks do not use 2-in-2-out elements, but switches that are more on the order of 20-in-20-out. This makes it possible for the number of levels in a network to be limited to 3 or 4. (The top level switches are known as *spine cards*.)

An extra complication to the analysis of networks in this case is the possibility of *oversubscription*. The ports in a network card are configurable as either in or out, and only the total is fixed. Thus, a 40-port switch can be configured as 20-in and 20-out, or 21-in and 19-out, et cetera. Of course, if all 21 nodes connected to the switch send at the same time, the 19 out ports will limit the bandwdidth.

There is a further problem. Let us considering building a small cluster with switches configured to have p in-ports and w out-ports, meaning we have $p+w$-port switches. Figure 2.32 depicts two of such switching,

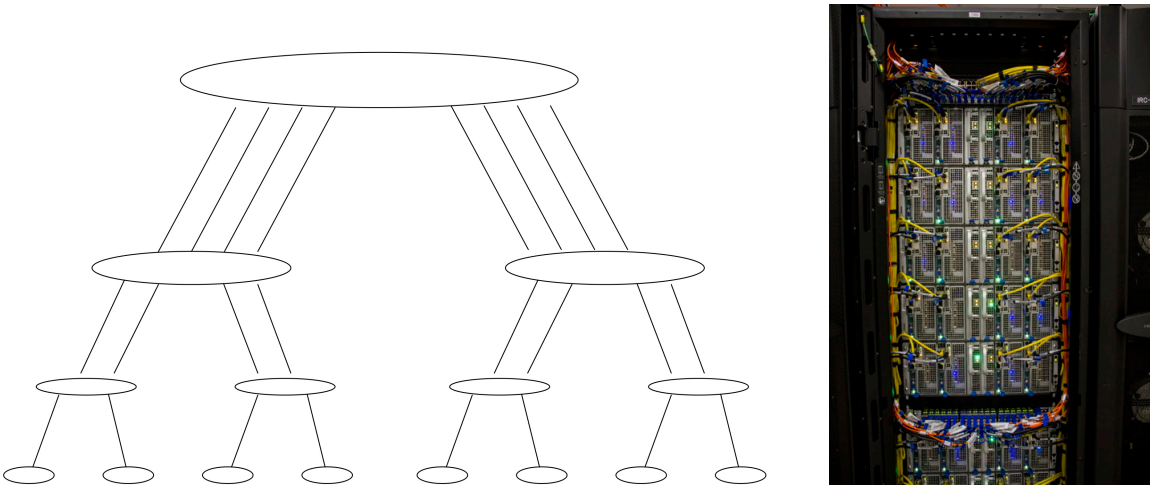


Figure 2.30: A fat tree with a three-level interconnect (left); the leaf switches in a cabinet of the Stampede cluster (right).

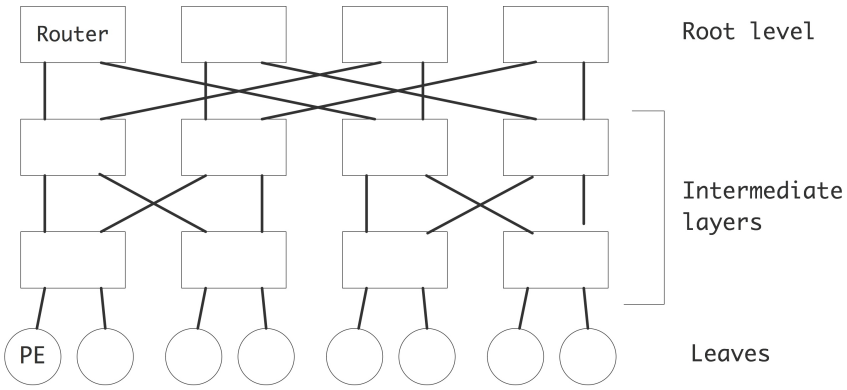


Figure 2.31: A fat-tree built from simple switching elements.

connecting a total of $2p$ nodes. If a node sends data through the switch, its choice of the w available wires is determined by the target node. This is known as *output routing*.

Clearly we can only expect w nodes to be able to send with message collisions, since that is the number of available wires between the switches. However, for many choices of w targets there will be contention for the wires regardless. This is an example of the *birthday paradox*.

Exercise 2.42. Consider the above architecture with p nodes sending through w wires between the switching. Code a simulation where $w' \leq w$ out of p nodes send a message to a randomly selected target node. What is the probability of collision as a function of w', w, p? Find a way to tabulate or plot data.
For bonus points, give a statistical analysis of the simple case $w' = 2$.

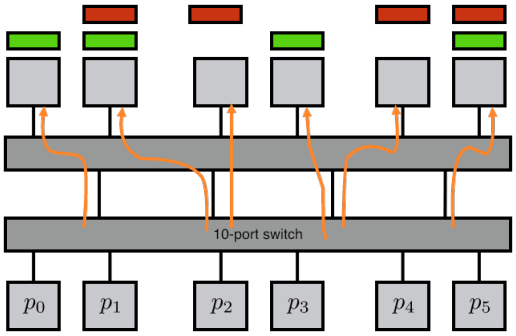


Figure 2.32: Port contention in an over-subscribed switch.

2.7.7 Cluster networks

The above discussion was somewhat abstract, but in real-life clusters you can actually see the network designs reflected. For instance, *fat tree cluster networks* will have a central cabinet corresponding to the top





Figure 2.33: Networks switches for the TACC Ranger and Stampede clusters.

level in the tree. Figure 2.33 shows the switches of the TACC *Ranger* (no longer in service) and *Stampede* clusters. In the second picture it can be seen that there are actually multiple redundant fat-tree networks.

On the other hand, clusters such as the *IBM BlueGene*, which is based on a *torus-based cluster*, will look like a collection of identical cabinets, since each contains an identical part of the network; see figure 2.34.



Figure 2.34: A BlueGene computer.

2.7.7.1 Case study: Stampede

As an example of networking in practice, let's consider the *Stampede* cluster at the Texas Advanced Computing Center. This can be described as a multi-root multi-stage fat-tree.

- Each rack consists of 2 chassis, with 20 nodes each.
- Each chassis has a leaf switch that is an internal *crossbar* that gives perfect connectivity between the nodes in a chassis;
- The leaf switch has 36 ports, with 20 connected to the nodes and 16 outbound. This *oversubscription* implies that at most 16 nodes can have perfect bandwidth when communicating outside the chassis.
- There are 8 central switches that function as 8 independent fat-tree root. Each chassis is connected by two connections to a 'leaf card' in each of the central switches, taking up precisely the 16 outbound ports.
- Each central switch has 18 spine cards with 36 ports each, with each port connecting to a different leaf card.
- Each central switch has 36 leaf cards with 18 ports to the leaf switches and 18 ports to the spine cards. This means we can support 648 chassis, of which 640 are actually used.

One of the optimizations in the network is that two connections to the same leaf card communicate without the latency of the higher tree levels. This means that 16 nodes in one chassis and 16 nodes in another can have perfect connectivity.

However, with *static routing*, such as used in *Infiniband*, there is a fixed port associated with each destination. (This mapping of destination to port is in the *routing tables* in each switch.) Thus, for some subsets of 16 nodes out 20 possible destination there will be perfect bandwidth, but other subsets will see the traffic for two destinations go through the same port.

2.7.7.2 Case study: Cray Dragonfly networks

The *Cray Dragonfly* network is an interesting practical compromise. Above we argued that a fully connected network would be too expensive to scale up. However, it is possible to have a fully connected set

of processors, if the number stays limited. The Dragonfly design uses small fully connected groups, and then makes a fully connected graph of these groups.

This introduces an obvious asymmetry, since processors inside a group have a larger bandwidth then between groups. However, thanks to *dynamic routing* messages can take a non-minimal path, being routed through other groups. This can alleviate the contention problem.

2.7.8 Bandwidth and latency

The statement above that sending a message can be considered a unit time operation, is of course unrealistic. A large message will take longer to transmit than a short one. There are two concepts to arrive at a more realistic description of the transmission process; we have already seen this in section 1.3.2 in the context of transferring data between cache levels of a processor.

latency Setting up a communication between two processors takes an amount of time that is independent of the message size. The time that this takes is known as the *latency* of a message. There are various causes for this delay.

- The two processors engage in 'hand-shaking', to make sure that the recipient is ready, and that appropriate buffer space is available for receiving the message.
- The message needs to be encoded for transmission by the sender, and decoded by the receiver.
- The actual transmission may take time: parallel computers are often big enough that, even at lightspeed, the first byte of a message can take hundreds of cycles to traverse the distance between two processors.

bandwidth After a transmission between two processors has been initiated, the main number of interest is the number of bytes per second that can go through the channel. This is known as the *bandwidth*. The bandwidth can usually be determined by the *channel rate*, the rate at which a physical link can deliver bits, and the *channel width*, the number of physical wires in a link. The channel width is typically a multiple of 16, usually 64 or 128. This is also expressed by saying that a channel can send one or two 8-byte words simultaneously.

Bandwidth and latency are formalized in the expression

$$T(n) = \alpha + \beta n$$

for the transmission time of an n-byte message. Here, α is the latency and β is the time per byte, that is, the inverse of bandwidth. Sometimes we consider data transfers that involve communication, for instance in the case of a *collective operation*; see section 7.1. We then extend the transmission time formula to

$$T(n) = \alpha + \beta n + \gamma n$$

where γ is the time per operation, that is, the inverse of the *computation rate*.

It would also be possible to refine this formulas as

$$T(n, p) = \alpha + \beta n + \delta p$$

where p is the number of network 'hops' that is traversed. However, on most networks the value of δ is far lower than of α, so we will ignore it here. Also, in fat-tree networks (section 2.7.6.3) the number

of hops is of the order of $\log P$, where P is the total number of processors, so it can never be very large anyway.

2.7.9 Locality in parallel computing

In section 1.6.2 you found a discussion of locality concepts in single-processor computing. The concept of *locality in parallel computing* comprises all this and a few levels more.

Between cores; private cache Cores on modern processors have private coherent caches. This means that it looks like you don't have to worry about locality, since data is accessible no matter what cache it is in. However, maintaining coherence costs bandwidth, so it is best to keep access localized.

Between cores; shared cache The cache that is shared between cores is one place where you don't have to worry about locality: this is memory that is truly symmetric between the processing cores.

Between sockets The sockets on a node (or motherboard) appear to the programmer to have shared memory, but this is really *NUMA* access (section 2.4.2) since the memory is associated with specific sockets.

Through the network structure Some networks have clear locality effects. You saw a simple example in section 2.7.1, and in general it is clear that any grid-type network will favor communication between 'nearby' processors. Networks based on fat-trees seem free of such contention issues, but the levels induce a different form of locality. One level higher than the locality on a node, small groups of nodes are often connected by a *leaf switch*, which prevents data from going to the central switch.

2.8 Multi-threaded architectures

The architecture of modern CPUs is largely dictated by the fact that getting data from memory is much slower than processing it. Hence, a hierarchy of ever faster and smaller memories tries to keep data as close to the processing unit as possible, mitigating the long latency and small bandwidth of main memory. The ILP in the processing unit also helps to hide the latency and more fully utilize the available bandwidth.

However, finding ILP is a job for the compiler and there is a limit to what it can practically find. On the other hand, scientific codes are often very *data parallel* in a sense that is obvious to the programmer, though not necessarily to the compiler. Would it be possible for the programmer to specify this parallelism explicitly and for the processor to use it?

In section 2.3.1 you saw that SIMD architectures can be programmed in an explicitly data parallel way. What if we have a great deal of data parallelism but not that many processing units? In that case, we could turn the parallel instruction streams into threads (see section 2.6.1) and have multiple threads be executed on each processing unit. Whenever a thread would stall because of an outstanding memory request, the processor could switch to another thread for which all the necessary inputs are available. This is called

multi-threading. While it sounds like a way of preventing the processor from waiting for memory, it can also be viewed as a way of keeping memory maximally occupied.

Exercise 2.43. If you consider the long latency and limited bandwidth of memory as two separate problems, does multi-threading address them both?

The problem here is that most CPUs are not good at switching quickly between threads. A *context switch* (switching between one thread and another) takes a large number of cycles, comparable to a wait for data from main memory. In a so-called *Multi-Threaded Architecture (MTA)* a context-switch is very efficient, sometimes as little as a single cycle, which makes it possible for one processor to work on many threads simultaneously.

The multi-threaded concept was explored in the *Tera Computer MTA* machine, which evolved into the current *Cray XMT*[7].

The other example of an MTA is the GPU, where the processors work as SIMD units, while being themselves multi-threaded; see section 2.9.3.

2.9 Co-processors, including GPUs

Current CPUs are built to be moderately efficient at just about any conceivable computation. This implies that by restricting the functionality of a processor it may be possible to raise its efficiency, or lower its power consumption at similar efficiency. Thus, the idea of incorporating a *co-processor* attached to the *host process* has been explored many times. For instance, Intel's 8086 chip, which powered the first generation of IBM PCs, could have a numerical co-processor, the *Intel 8087 co-processor*, added to it. This co-processor was very efficient at transcendental functions and it also incorporated SIMD technology. Using separate functionality for graphics has also been popular, leading to the *SSE* instructions for the x86 processor, and separate GPU units to be attached to the PCI-X bus.

Further examples are the use of co-processors for Digital Signal Processing (DSP) instructions, as well as FPGA boards which can be reconfigured to accommodate specific needs. Early *array processors* such as the *ICL DAP* were also co-processors.

In this section we look briefly at some modern incarnations of this idea, in particular GPUs.

2.9.1 A little history

Co-processors can be programmed in two different ways: sometimes it is seamlessly integrated, and certain instructions are automatically executed there, rather than on the 'host' processor. On the other hand, it is also possible that co-processor functions need to be explicitly invoked, and it may even be possible to overlap co-processor functions with host functions. The latter case may sound attractive from an efficiency point of view, but it raises a serious problem of programmability. The programmer now needs to identify explicitly two streams of work: one for the host processor and one for the co-processor.

Some notable parallel machines with co-processors are:

7. Tera Computer renamed itself *Cray Inc.* after acquiring *Cray Research* from *SGI*.

- The *Intel Paragon* (1993) had two processors per node, one for communication and the other for computation. These were in fact identical, the *Intel i860*Intel i860 processor. In a later revision, it became possible to pass data and function pointers to the communication processors.
- The *IBM Roadrunner* at Los Alamos was the first machine to reach a PetaFlop. (The *Grape computer* had reached this point earlier, but that was a special purpose machine for molecular dynamics calculations.) It achieved this speed through the use of Cell co-processors. Incidentally, the Cell processor is in essence the engine of the Sony Playstation3, showing again the commoditization of supercomputers (section 2.3.3).
- The Chinese *Tianhe-1A* topped the Top 500 list in 2010, reaching about 2.5 PetaFlops through the use of NVidia GPUs.
- The *Tianhe-2* and the *TACC Stampede cluster* use *Intel Xeon Phi* co-processors.

The Roadrunner and Tianhe-1A are examples of co-processors that are very powerful, and that need to be explicitly programmed independently of the host CPU. For instance, code running on the GPUs of the Tianhe-1A is programmed in *CUDA* and compiled separately.

In both cases the programmability problem is further exacerbated by the fact that the co-processor can not directly talk to the network. To send data from one co-processor to another it has to be passed to a host processor, from there through the network to the other host processor, and only then moved to the target co-processor.

2.9.2 Bottlenecks

Co-processors often have their own memory, and the *Intel Xeon Phi* can run programs independently, but more often there is the question of how to access the memory of the host processor. A popular solution is to connect the co-processor through a *PCI bus*. Accessing host memory this way is slower than the direct connection that the host processor has. For instance, the *Intel Xeon Phi* has a *bandwidth* of 512-bit wide at 5.5GT per second (we will get to that 'GT' in a second), while its connection to host memory is 5.0GT/s, but only 16-bit wide.

GT measure We are used to seeing bandwidth measured in gigabits per second. For a *PCI bus* one often see the *GT* measure. This stands for giga-transfer, and it measures how fast the bus can change state between zero and one. Normally, every state transition would correspond to a bit, except that the bus has to provide its own clock information, and if you would send a stream of identical bits the clock would get confused. Therefore, every 8 bits are encoded in 10 bits, to prevent such streams. However, this means that the effective bandwidth is lower than the theoretical number, by a factor of $4/5$ in this case.

And since manufacturers like to give a positive spin on things, they report the higher number.

2.9.3 GPU computing

A *Graphics Processing Unit (GPU)* (or sometimes *General Purpose Graphics Processing Unit (GPGPU)*) is a special purpose processor, designed for fast graphics processing. However, since the operations done

for graphics are a form of arithmetic, GPUs have gradually evolved a design that is also useful for non-graphics computing. The general design of a GPU is motivated by the 'graphics pipeline': identical operations are performed on many data elements in a form of *data parallelism* (section 2.5.1), and a number of such blocks of data parallelism can be active at the same time.

The basic limitations that hold for a CPU hold for a GPU: accesses to memory incur a long latency. The solution to this problem in a CPU is to introduce levels of cache; in the case of a GPU a different approach is taken (see also section 2.8). GPUs are concerned with *throughput computing*, delivering large amounts of data with high average rates, rather than any single result as quickly as possible. This is made possible by supporting many threads (section 2.6.1) and having very fast switching between them. While one thread is waiting for data from memory, another thread that already has its data can proceed with its computations.

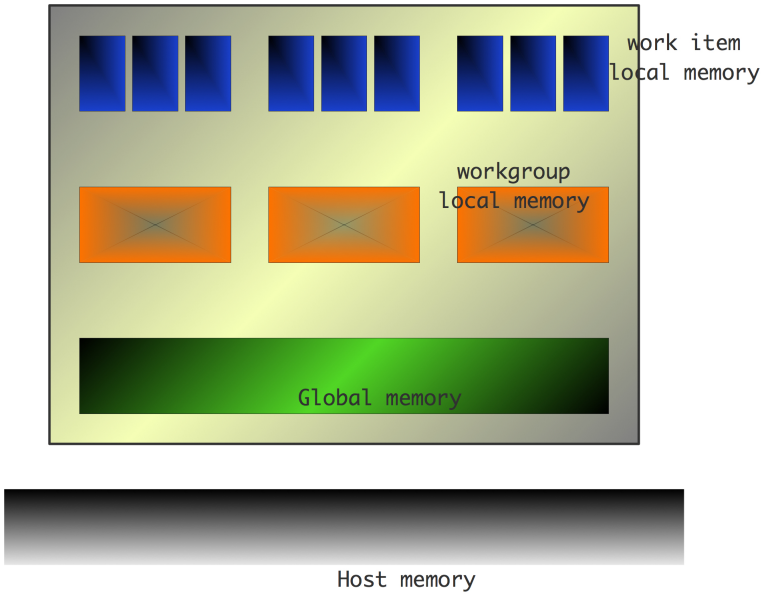


Figure 2.35: Memory structure of a GPU.

2.9.3.1 *SIMD-type programming with kernels*

Present day GPUs[8] have an architecture that combines SIMD and SPMD parallelism. Threads are not completely independent, but are ordered in *thread blocks*, where all threads in the block execute the same instruction, making the execution SIMD. It is also possible to schedule the same instruction stream (a 'kernel' in Cuda terminology) on more than one thread block. In this case, thread blocks can be out of sync, much like processes in an SPMD context. However, since we are dealing with threads here, rather than processes, the term *Single Instruction Multiple Thread (SIMT)* is used.

This software design is apparent in the hardware; for instance, an NVidia GPU has 16–30 Streaming Multiprocessors (SMs), and a SMs consists of 8 Streaming Processors (SPs), which correspond to processor cores; see figure 2.36. The SPs act in true SIMD fashion. The number of cores in a GPU is typically larger than in traditional multi-core processors, but the cores are more limited. Hence, the term *manycore* is used here.

The SIMD, or *data parallel*, nature of GPUs becomes apparent in the way *CUDA* starts processes. A *kernel*, that is, a function that will be executed on the GPU, is started on *mn* cores by:

```
KernelProc<< m,n >>(args)
```

8. The most popular GPUs today are made by NVidia, and are programmed in *CUDA*, an extension of the C language.

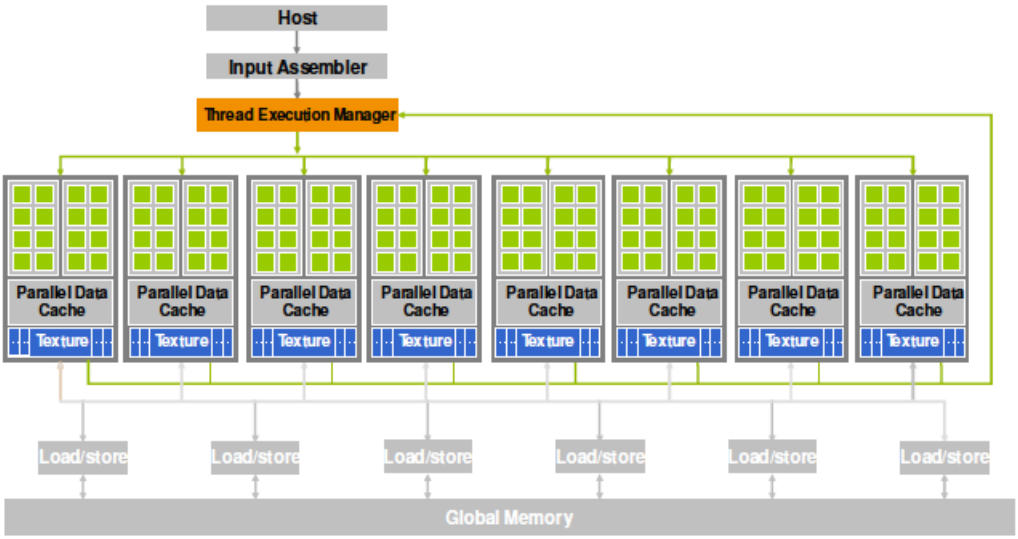


Figure 2.36: Diagram of a GPU.

The collection of mn cores executing the kernel is known as a *grid*, and it is structured as m *thread blocks* of n threads each. A thread block can have up to 512 threads.

Recall that threads share an address space (see section 2.6.1), so they need a way to identify what part of the data each thread will operate on. For this, the blocks in a thread are numbered with x, y coordinates, and the threads in a block are numbered with x, y, z coordinates. Each thread knows its coordinates in the block, and its block's coordinates in the grid.

We illustrate this with a vector addition example:

```
// Each thread performs one addition
__global__ void vecAdd(float* A, float* B, float* C)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  C[i] = A[i] + B[i];
}
int main()
{
  // Run grid of N/256 blocks of 256 threads each
  vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

This shows the SIMD nature of GPUs: every thread executes the same scalar program, just on different data.

Threads in a thread block are truly data parallel: if there is a conditional that makes some threads take the *true* branch and others the *false* branch, then one branch will be executed first, with all threads in the

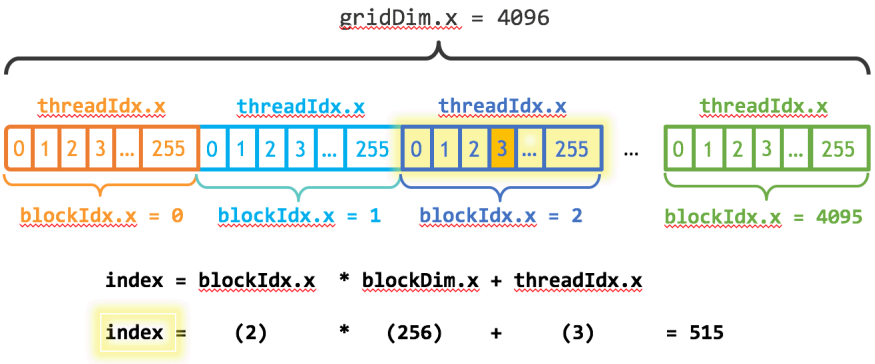


Figure 2.37: Thread indexing in CUDA.

other branch stopped. Subsequently, *and not simultaneously*, the threads on the other branch will then execute their code. This may induce a severe performance penalty.

GPUs rely on a large amount of data parallelism and the ability to do a fast *context switch*. This means that they will thrive in graphics and scientific applications, where there is lots of data parallelism. However they are unlikely to do well on 'business applications' and operating systems, where the parallelism is of the Instruction Level Parallelism (ILP) type, which is usually limited.

2.9.3.2 GPUs versus CPUs

These are some of the differences between GPUs and regular CPUs:

- First of all, as of this writing (late 2010), GPUs are attached processors, for instance over a *PCI-X bus*, so any data they operate on has to be transferred from the CPU. Since the memory *bandwidth* of this transfer is low, at least 10 times lower than the memory bandwidth in the GPU, sufficient work has to be done on the GPU to overcome this overhead.
- Since GPUs are graphics processors, they put an emphasis on *single precision* floating point arithmetic. To accommodate the scientific computing community, *double precision* support is increasing, but double precision speed is typically half the single precision flop rate. This discrepancy is likely to be addressed in future generations.
- A CPU is optimized to handle a single stream of instructions that can be very heterogeneous in character; a GPU is made explicitly for data parallelism, and will perform badly on traditional codes.
- A CPU is made to handle one *thread*, or at best a small number of threads. A GPU *needs* a large number of threads, far larger than the number of computational cores, to perform efficiently.

2.9.3.3 Expected benefit from GPUs

GPUs have rapidly gained a reputation for achieving high performance, highly cost effectively. Stories abound of codes that were ported with minimal effort to CUDA, with a resulting speedup of sometimes 400 times. Is the GPU really such a miracle machine? Were the original codes badly programmed? Why don't we use GPUs for everything if they are so great?

The truth has several aspects.

First of all, a GPU is not as general-purpose as a regular CPU: GPUs are very good at doing *data parallel* computing, and CUDA is good at expressing this fine-grained parallelism elegantly. In other words, GPUs are suitable for a certain type of computation, and will be a poor fit for many others.

Conversely, a regular CPU is not necessarily good at data parallelism. Unless the code is very carefully written, performance can degrade from optimal by approximately the following factors:

- Unless directives or explicit parallel constructs are used, compiled code will only use 1 out of the available cores, say 4.
- If instructions are not pipelined, the latency because of the floating point pipeline adds another factor of 4.
- If the core has independent add and multiply pipelines, another factor of 2 will be lost if they are not both used simultaneously.
- Failure to use SIMD registers can add more to the slowdown with respect to peak performance.

Writing the optimal CPU implementation of a computational kernel often requires writing in assembler, whereas straightforward CUDA code will achieve high performance with comparatively little effort, provided of course the computation has enough data parallelism.

2.9.4 Intel Xeon Phi

Intel *Xeon Phi* (also known by its architecture design as Many Integrated Cores (MIC)) is a design expressly designed for numerical computing. The initial design, the *Knights Corner* was a co-processor, though the second iteration, the *Knights Landing* was self-hosted.

As a co-processor, the Xeon Phi has both differences and similarities with GPUs.

- Both are connected through a *PCI-X* bus, which means that operations on the device have a considerable latency in starting up.
- The Xeon Phi has general purpose cores, so that it can run whole programs; GPUs has that only to a limited extent (see section 2.9.3.1).
- The Xeon Phi accepts ordinary C code.
- Both architectures require a large amount of SIMD-style parallelism, in the case of the Xeon Phi because of the 8-word wide *AVX* instructions.
- Both devices work, or can work, through *offloading* from a host program. In the case of the Xeon Phi this can happen with OpenMP constructs and *MKL* calls.

2.10 Load balancing

In much of this chapter, we assumed that a problem could be perfectly divided over processors, that is, a processor would always be performing useful work, and only be *idle* because of latency in communication. In practice, however, a processor may be idle because it is waiting for a message, and the sending processor has not even reached the send instruction in its code. Such a situation, where one processor is working and another is idle, is described as *load unbalance*: there is no intrinsic reason for the one processor to be idle, and it could have been working if we had distributed the work load differently.

There is an asymmetry between processors having too much work and having not enough work: it is better to have one processor that finishes a task early, than having one that is overloaded so that all others wait for it.

Exercise 2.44. Make this notion precise. Suppose a parallel task takes time 1 on all processors but one.

- Let $0 < \alpha < 1$ and let one processor take time $1 + \alpha$. What is the speedup and efficiency as function of the number of processors? Consider this both in the Amdahl and Gustafsson sense (section 2.2.3).
- Answer the same questions if one processor takes time $1 - \alpha$.

Load balancing is often expensive since it requires moving relatively large amounts of data. For instance, section 7.5 has an analysis showing that the data exchanges during a sparse matrix-vector product is of a lower order than what is stored on the processor. However, we will not go into the actual cost of moving: our main concerns here are to balance the workload, and to preserve any locality in the original load distribution.

2.10.1 Load balancing versus data distribution

There is a duality between work and data: in many applications the distribution of data implies a distribution of work and the other way around. If an application updates a large array, each element of the array typically 'lives' on a uniquely determined processor, and that processor takes care of all the updates to that element. This strategy is known as *owner computes*.

Thus, there is a direct relation between data and work, and, correspondingly, data distribution and load balancing go hand in hand. For instance, in section 7.2 we will talk about how data distribution influences the efficiency, but this immediately translates to concerns about load distribution:

- Load needs to be evenly distributed. This can often be done by evenly distributing the data, but sometimes this relation is not linear.
- Tasks need to be placed to minimize the amount of traffic between them. In the matrix-vector multiplication case this means that a two-dimensional distribution is to be preferred over a one-dimensional one; the discussion about *space-filling curves* is similarly motivated.

As a simple example of how the data distribution influences the load balance, consider a linear array where each point undergoes the same computation, and each computation takes the same amount of time. If the length of the array, N, is perfectly divisible by the number of processors, p, the work is perfectly evenly distributed. If the data is not evenly divisible, we start by assigning $\lfloor N/p \rfloor$ points to each processor, and the remaining $N - p\lfloor N/p \rfloor$ points to the last processors.

Exercise 2.45. In the worst case, how unbalanced does that make the processors' work? Compare this scheme to the option of assigning $\lceil N/p \rceil$ points to all processors except one, which gets fewer; see the exercise above.

It is better to spread the surplus $r = N - p\lfloor N/p \rfloor$ over r processors than one. This could be done by giving one extra data point to the first or last r processors. This can be achieved by assigning to process p the range

$$\big[p \times \lfloor (N + p - 1)/p \rfloor, (p + 1) \times \lfloor (N + p - 1)/p \rfloor\big)$$

While this scheme is decently balanced, computing for instance to what processor a given point belongs is tricky. The following scheme makes such computations easier: let $f(i) = \lfloor iN/p \rfloor$, then processor i gets points $f(i)$ up to $f(i+1)$.

Exercise 2.46. Show that $\lfloor N/p \rfloor \leq f(i+1) - f(i) \leq \lceil N/p \rceil$.

Under this scheme, the processor that owns index i is $\lfloor (p(i+1)-1)/N \rfloor$.

2.10.2 Load scheduling

In some circumstances, the computational load can be freely assigned to processors, for instance in the context of shared memory where all processors have access to all the data. In that case we can consider the difference between *static scheduling* using a pre-determined assignment of work to processors, or *dynamic scheduling* where the assignment is determined during executions.

As an illustration of the merits of dynamic scheduling consider scheduling 8 tasks of decreasing runtime on 4 threads (figure 2.38). In static scheduling, the first thread gets tasks 1 and 4, the second 2 and 5, et

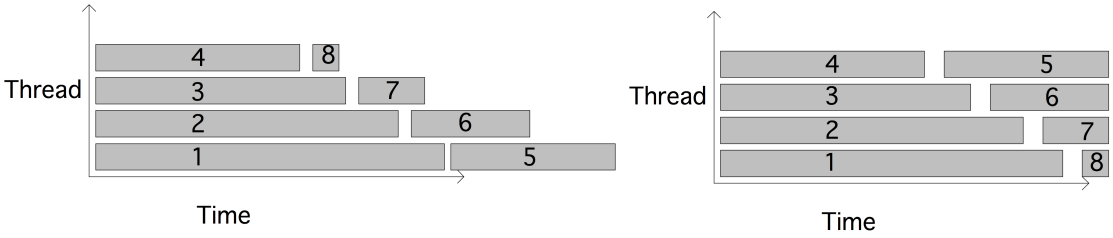


Figure 2.38: Static or round-robin (left) vs dynamic (right) thread scheduling; the task numbers are indicated.

cetera. In dynamic scheduling, any thread that finishes its task gets the next task. This clearly gives a better running time in this particular example. On the other hand, dynamic scheduling is likely to have a higher overhead.

2.10.3 Load balancing of independent tasks

In other cases, work load is not directly determined by data. This can happen if there is a pool of work to be done, and the processing time for each work item is not easily computed from its description. In such cases we may want some flexibility in assigning work to processes.

Let us first consider the case of a job that can be partitioned into independent tasks that do not communicate. An example would be computing the pixels of a *Mandelbrot set* picture, where each pixel is set according to a mathematical function that does not depend on surrounding pixels. If we could predict the time it would take to draw an arbitrary part of the picture, we could make a perfect division of the work and assign it to the processors. This is known as *static load balancing*.

More realistically, we can not predict the running time of a part of the job perfectly, and we use an *overdecomposition* of the work: we divide the work in more tasks than there are processors. These tasks are then assigned to a *work pool*, and processors take the next job from the pool whenever they finish a

job. This is known as *dynamic load balancing*. Many graph and combinatorial problems can be approached this way; see section 2.5.3. For task assignment in a multicore context, see section 7.12.

There are results that show that randomized assignment of tasks to processors is statistically close to optimal [119], but this ignores the aspect that in scientific computing tasks typically communicate frequently.

Exercise 2.47. Suppose you have tasks $\{T_i\}_{i=1,\ldots,N}$ with running times t_i, and an unlimited number of processors. Look up *Brent's theorem* in section 2.2.4, and derive from it that the fastest execution scheme for the tasks can be characterized as follows: there is one processor that only executes the task with maximal t_i value. (This exercise was inspired by [164].)

2.10.4 Load balancing as graph problem

Next let us consider a parallel job where the parts do communicate. In this case we need to balance both the scalar workload and the communication.

A parallel computation can be formulated as a graph (see Appendix 20 for an introduction to graph theory) where the processors are the vertices, and there is an edge between two vertices if their processors need to communicate at some point. Such a graph is often derived from an underlying graph of the problem being solved. As an example consider the matrix-vector product $y = Ax$ where A is a sparse matrix, and look in detail at the processor that is computing y_i for some i. The statement $y_i \leftarrow y_i + A_{ij}x_j$ implies that this processor will need the value x_j, so, if this variable is on a different processor, it needs to be sent over.

We can formalize this: Let the vectors x and y be distributed disjointly over the processors, and define uniquely $P(i)$ as the processor that owns index i. Then there is an edge (P, Q) if there is a nonzero element a_{ij} with $P = P(i)$ and $Q = P(j)$. This graph is undirected in the case of a *structurally symmetric matrix*, that is $a_{ij} \neq 0 \Leftrightarrow a_{ji} \neq 0$.

The distribution of indices over the processors now gives us vertex and edge weights: a processor has a vertex weight that is the number of indices owned by it; an edge (P, Q) has a weight that is the number of vector components that need to be sent from Q to P, as described above.

The load balancing problem can now be formulated as follows:

> Find a partitioning $\mathbb{P} = \cup_i \mathbb{P}_i$, such the variation in vertex weights is minimal, and simultaneously the edge weights are as low as possible.

Minimizing the variety in vertex weights implies that all processor have approximately the same amount of work. Keeping the edge weights low means that the amount of communication is low. These two objectives need not be satisfiable at the same time: some trade-off is likely.

Exercise 2.48. Consider the limit case where processors are infinitely fast and bandwidth between processors is also unlimited. What is the sole remaining factor determining the runtime? What graph problem do you need to solve now to find the optimal load balance? What property of a sparse matrix gives the worst case behavior?

An interesting approach to load balancing comes from spectral graph theory (section 20.6): if A_G is the adjacency matrix of an undirected graph and $D_G - A_G$ the *graph Laplacian*, then the eigenvector u_1 to the smallest eigenvalue zero is positive, and the eigenvector u_2 to the next eigenvalue is orthogonal to

it. Therefore u_2 has to have elements of alternating sign; further analysis shows that the elements with positive sign are connected, as are the negative ones. This leads to a natural bisection of the graph.

2.10.5 Load redistributing

In certain applications an initial load distribution is clear, but later adjustments are needed. A typical example is in Finite Element Method (FEM) codes, where load can be distributed by a partitioning of the physical domain; see section 7.5.3. If later the discretization of the domain changes, the load has to be *rebalanced* or *redistributed*. In the next subsections we will see techniques for load balancing and rebalancing aimed at preserving locality.

2.10.5.1 Diffusion load balancing

In many practical situations we can associate a *processor graph* with our problem: there is a vertex between any pair of processes that directly interacts through point-to-point communication. Thus, it seems a natural thought to use this graph in load balancing, and only move load from a processor to its neighbors in the graph.

This is the idea by *diffusion* load balancing [38, 109].

While the graph is not intrinsically directed, for load balancing we put arbitrary directions on the edges. Load balancing is then described as follows.

Let ℓ_i be the load on process i, and $\tau_i^{(j)}$ the transfer of load on an edge $j \to i$. Then

$$\ell_i \leftarrow \ell_i + \sum_{j\to i} \tau_i^{(j)} - \sum_{i\to j} \tau_j^{(i)}$$

Although we just used a i,j number of edges, in practice we put a linear numbering the edges. We then get a system

$$AT = \bar{L}$$

where

- A is a matrix of size $|N| \times |E|$ describing what edges connect in/out of a node, with elements values equal to ± 1 depending;
- T is the vector of transfers, of size $|E|$; and
- $\bar{L}$ is the load deviation vector, indicating for each node how far over/under the average load they are.

In the case of a linear processor array this matrix is under-determined, with fewer edges than processors, but in most cases the system will be over-determined, with more edges than processes. Consequently, we solve

$$T = (A^tA)^{-1}A^t\bar{L} \qquad \text{or} T = A^t(AA^t)^{-1}\bar{L}.$$

Since A^tA and AA^t are positive indefinite, we could solve the approximately by relaxation, needing only local knowledge. Of course, such relaxation has slow convergence, and a global method, such as Conjugate Gradients (CG), would be faster [109].

2.10.5.2 *Load balancing with space-filling curves*

In the previous sections we considered two aspects of load balancing: making sure all processors have an approximately equal amount of work, and letting the distribution reflect the structure of the problem so that communication is kept within reason. We can phrase the second point trying to preserve the locality of the problem when distributed over a parallel machine: points in space that are close together are likely to interact, so they should be on the same processor, or at least one not too far away.

Striving to preserve locality is not obviously the right strategy. In BSP (see section 2.6.8) a statistical argument is made that *random placement* will give a good load balance as well as balance of communication.

Exercise 2.49. Consider the assignment of processes to processors, where the structure of the problem is such that each processes only communicates with its nearest neighbors, and let processors be ordered in a two-dimensional grid. If we do the obvious assignment of the process grid to the processor grid, there will be no contention. Now write a program that assigns processes to random processors, and evaluate how much contention there will be.

In the previous section you saw how graph partitioning techniques can help with the second point of preserving problem locality. In this section you will see a different technique that is attractive both for the initial load assignment and for subsequent *load rebalancing*. In the latter case, a processor's work may increase or decrease, necessitating moving some of the load to a different processor.

For instance, some problems are adaptively refined[9] . This is illustrated in figure 2.39. If we keep track

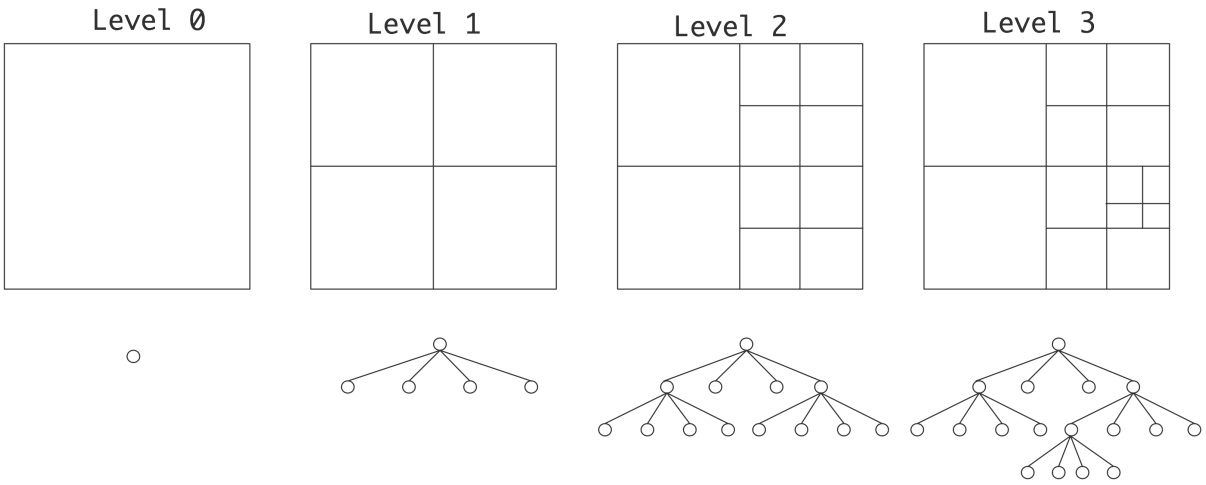


Figure 2.39: Adaptive refinement of a domain in subsequent levels.

of these refinement levels, the problem gets a tree structure, where the leaves contain all the work. Load balancing becomes a matter of partitioning the leaves of the tree over the processors; figure 2.40. Now we observe that the problem has a certain locality: the subtrees of any non-leaf node are physically close, so there will probably be communication between them.

9. For a detailed discussion, see [28].

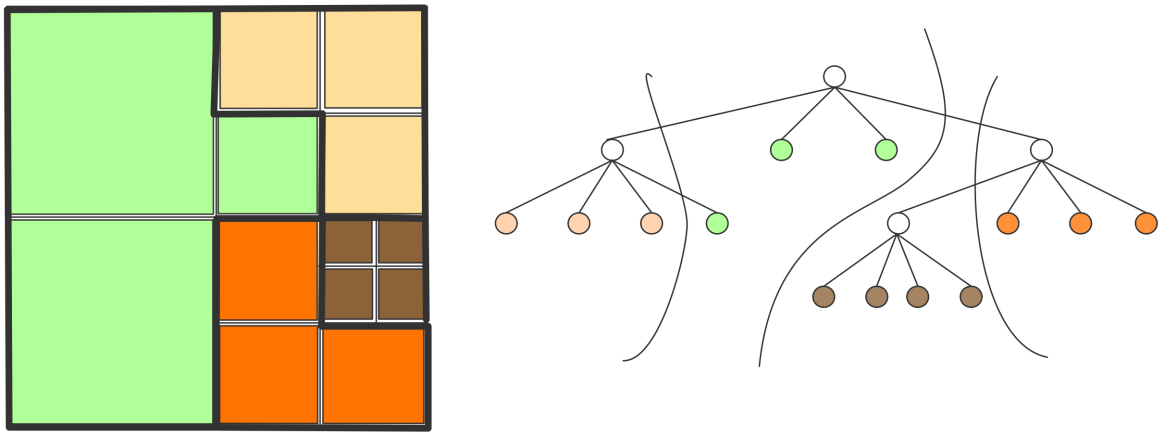


Figure 2.40: Load distribution of an adaptively refined domain.

- Likely there will be more subdomains than processors; to minimize communication between processors, we want each processor to contain a simply connected group of subdomains. Moreover, we want each processor to cover a part of the domain that is 'compact' in the sense that it has low aspect ratio, and low surface-to-volume ratio.
- When a subdomain gets further subdivided, part of the load of its processor may need to be shifted to another processor. This process of *load redistributing* should preserve locality.

To fulfill these requirements we use Space-Filling Curves (SFCs). A Space-Filling Curve (SFC) for the load balanced tree is shown in figure 2.41. We will not give a formal discussion of SFCs; instead we will let

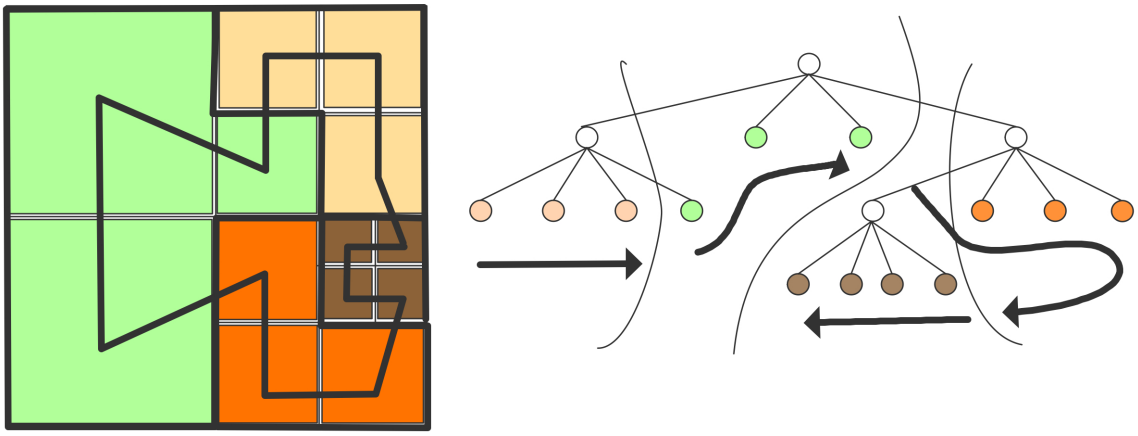


Figure 2.41: A space filling curve for the load balanced tree.

figure 2.42 stand for a definition: a SFC is a recursively defined curve that touches each subdomain once[10] . The SFC has the property that domain elements that are close together physically will be close together

10. Space-Filling Curves (SFCs) were introduced by Peano as a mathematical device for constructing a continuous surjective function from the line segment $[0,1]$ to a higher dimensional cube $[0,1]^d$. This upset the intuitive notion of dimension that 'you

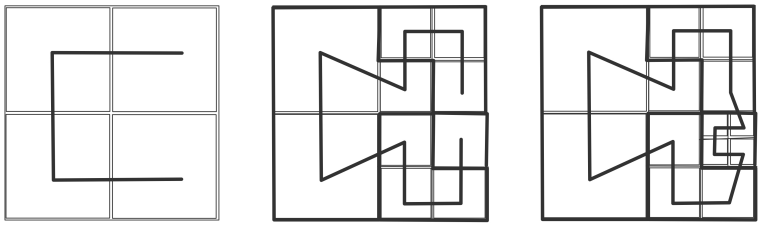


Figure 2.42: Space filling curves, regularly and irregularly refined.

on the curve, so if we map the SFC to a linear ordering of processors we will preserve the locality of the problem.

More importantly, if the domain is refined by another level, we can refine the curve accordingly. Load can then be redistributed to neighboring processors on the curve, and we will still have locality preserved.

(The use of Space-Filling Curves (SFCs) is N-body problems was discussed in [193] and [180].)

2.11 Remaining topics

2.11.1 Distributed computing, grid computing, cloud computing

In this section we will take a short look at terms such as *cloud computing*, and an earlier term *distributed computing*. These are concepts that have a relation to parallel computing in the scientific sense, but that differ in certain fundamental ways.

Distributed computing can be traced back as coming from large database servers, such as airline reservations systems, which had to be accessed by many travel agents simultaneously. For a large enough volume of database accesses a single server will not suffice, so the mechanism of *remote procedure call* was invented, where the central server would call code (the procedure in question) on a different (remote) machine. The remote call could involve transfer of data, the data could be already on the remote machine, or there would be some mechanism that data on the two machines would stay synchronized. This gave rise to the *Storage Area Network (SAN)*. A generation later than distributed database systems, web servers had to deal with the same problem of many simultaneous accesses to what had to act like a single server.

We already see one big difference between distributed computing and high performance parallel computing. Scientific computing needs parallelism because a single simulation becomes too big or slow for one machine; the business applications sketched above deal with many users executing small programs (that is, database or web queries) against a large data set. For scientific needs, the processors of a parallel machine (the nodes in a cluster) have to have a very fast connection to each other; for business needs no such network is needed, as long as the central dataset stays coherent.

Both in *HPC* and in business computing, the server has to stay available and operative, but in distributed computing there is considerably more liberty in how to realize this. For a user connecting to a service

---

can not stretch and fold a line segment to fill up the square'. A proper treatment of the concept of dimension was later given by Brouwer.

such as a database, it does not matter what actual server executes their request. Therefore, distributed computing can make use of *virtualization*: a virtual server can be spawned off on any piece of hardware.

An analogy can be made between remote servers, which supply computing power wherever it is needed, and the electric grid, which supplies electric power wherever it is needed. This has led to *grid computing* or *utility computing*, with the Teragrid, owned by the US National Science Foundation, as an example. Grid computing was originally intended as a way of hooking up computers connected by a *Local Area Network (LAN)* or *Wide Area Network (WAN)*, often the Internet. The machines could be parallel themselves, and were often owned by different institutions. More recently, it has been viewed as a way of sharing resources, both datasets, software resources, and scientific instruments, over the network.

The notion of utility computing as a way of making services available, which you recognize from the above description of distributed computing, went mainstream with Google's search engine, which indexes the whole of the Internet. Another example is the GPS capability of Android mobile phones, which combines GIS, GPS, and mashup data. The computing model by which Google's gathers and processes data has been formalized in MapReduce [41]. It combines a data parallel aspect (the 'map' part) and a central accumulation part ('reduce'). Neither involves the tightly coupled neighbor-to-neighbor communication that is common in scientific computing. An open source framework for MapReduce computing exists in Hadoop [93]. Amazon offers a commercial Hadoop service.

The concept of having a remote computer serve user needs is attractive even if no large datasets are involved, since it absolves the user from the need of maintaining software on their local machine. Thus, Google Docs offers various 'office' applications without the user actually installing any software. This idea is sometimes called *Software As a Service (SAS)*, where the user connects to an 'application server', and accesses it through a client such as a web browser. In the case of Google Docs, there is no longer a large central dataset, but each user interacts with their own data, maintained on Google's servers. This of course has the large advantage that the data is available from anywhere the user has access to a web browser.

The SAS concept has several connections to earlier technologies. For instance, after the mainframe and workstation eras, the so-called *thin client* idea was briefly popular. Here, the user would have a workstation rather than a terminal, yet work on data stored on a central server. One product along these lines was Sun's *Sun Ray* (circa 1999) where users relied on a smartcard to establish their local environment on an arbitrary, otherwise stateless, workstation.

2.11.1.1 Usage scenarios

The model where services are available on demand is attractive for businesses, which increasingly are using cloud services. The advantages are that it requires no initial monetary and time investment, and that no decisions about type and size of equipment have to be made. At the moment, cloud services are mostly focused on databases and office applications, but scientific clouds with a high performance interconnect are under development.

The following is a broad classification of usage scenarios of cloud resources[11] .

11. Based on a blog post by Ricky Ho: http://blogs.globallogic.com/five-cloud-computing-patterns.

- Scaling. Here the cloud resources are used as a platform that can be expanded based on user demand. This can be considered Platform-as-a-Service (PAS): the cloud provides software and development platforms, eliminating the administration and maintenance for the user.
  We can distinguish between two cases: if the user is running single jobs and is actively waiting for the output, resources can be added to minimize the wait time for these jobs (capability computing). On the other hand, if the user is submitting jobs to a queue and the time-to-completion of any given job is not crucial (capacity computing), resources can be added as the queue grows.
  In HPC applications, users can consider the cloud resources as a cluster; this falls under Infrastructure-as-a-Service (IAS): the cloud service is a computing platforms allowing customization at the operating system level.
- Multi-tenancy. Here the same software is offered to multiple users, giving each the opportunity for individual customizations. This falls under Software-as-a-Service (SAS): software is provided on demand; the customer does not purchase software, but only pays for its use.
- Batch processing. This is a limited version of one of the Scaling scenarios above: the user has a large amount of data to process in batch mode. The cloud then becomes a batch processor. This model is a good candidate for MapReduce computations; section 2.11.3.
- Storage. Most cloud providers offer database services, so this model absolves the user from maintaining their own database, just like the Scaling and Batch processing models take away the user's concern with maintaining cluster hardware.
- Synchronization. This model is popular for commercial user applications. Netflix and Amazon's Kindle allow users to consume online content (streaming movies and ebooks respectively); after pausing the content they can resume from any other platform. Apple's recent iCloud provides synchronization for data in office applications, but unlike Google Docs the applications are not 'in the cloud' but on the user machine.

The first Cloud to be publicly accessible was Amazon's Elastic Compute cloud (EC2), launched in 2006. EC2 offers a variety of different computing platforms and storage facilities. Nowadays more than a hundred companies provide cloud based services, well beyond the initial concept of computers-for-rent.

The infrastructure for cloud computing can be interesting from a computer science point of view, involving distributed file systems, scheduling, virtualization, and mechanisms for ensuring high reliability.

An interesting project, combining aspects of grid and cloud computing is the Canadian Advanced Network For Astronomical Research[172]. Here large central datasets are being made available to astronomers as in a grid, together with compute resources to perform analysis on them, in a cloud-like manner. Interestingly, the cloud resources even take the form of user-configurable virtual clusters.

2.11.1.2 Characterization

Summarizing[12] we have three *cloud computing service models*:

Software as a Service The consumer runs the provider's application, typically through a thin client such as a browser; the consumer does not install or administer software. A good example is Google Docs

12. The remainder of this section is based on the NIST definition of cloud computing http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

Platform as a Service The service offered to the consumer is the capability to run applications developed by the consumer, who does not otherwise manage the processing platform or data storage involved.

Infrastructure as a Service The provider offers to the consumer both the capability to run software, and to manage storage and networks. The consumer can be in charge of operating system choice and network components such as firewalls.

These can be deployed as follows:

Private cloud The cloud infrastructure is managed by one organization for its own exclusive use.

Public cloud The cloud infrastructure is managed for use by a broad customer base.

One could also define hybrid models such as community clouds.

The characteristics of cloud computing are then:

On-demand and self service The consumer can quickly request services and change service levels, without requiring human interaction with the provider.

Rapid elasticity The amount of storage or computing power appears to the consumer to be unlimited, subject only to budgeting constraints. Requesting extra facilities is fast, in some cases automatic.

Resource pooling Virtualization mechanisms make a cloud appear like a single entity, regardless its underlying infrastructure. In some cases the cloud remembers the 'state' of user access; for instance, Amazon's Kindle books allow one to read the same book on a PC, and a smartphone; the cloud-stored book 'remembers' where the reader left off, regardless the platform.

Network access Clouds are available through a variety of network mechanisms, from web browsers to dedicated portals.

Measured service Cloud services are typically 'metered', with the consumer paying for computing time, storage, and bandwidth.

2.11.2 Capability versus capacity computing

Large parallel computers can be used in two different ways. In later chapters you will see how scientific problems can be scaled up almost arbitrarily. This means that with an increasing need for accuracy or scale, increasingly large computers are needed. The use of a whole machine for a single problem, with only time-to-solution as the measure of success, is known as *capability computing*.

On the other hand, many problems need less than a whole supercomputer to solve, so typically a computing center will set up a machine so that it serves a continuous stream of user problems, each smaller than the full machine. In this mode, the measure of success is the sustained performance per unit cost. This is known as *capacity computing*, and it requires a finely tuned *job scheduling* strategy.

A popular scheme is *fair-share scheduling*, which tries to allocate resources equally between users, rather than between processes. This means that it will lower a user's priority if the user had recent jobs, and it will give higher priority to short or small jobs. Examples of schedulers on this principle are *SGE* and *Slurm*.

Jobs can have dependencies, which makes scheduling harder. In fact, under many realistic conditions scheduling problems are *NP-complete*, so in practice heuristics will be used. This topic, while interesting, is not further discussed in this book.

2.11.3 MapReduce

MapReduce [41] is a programming model for certain parallel operations. One of its distinguishing characteristics is that it is implemented using *functional programming*. The MapReduce model handles computations of the following form:

- For all available data, select items that satisfy a certain criterion;
- and emit a key-value pair for them. This is the mapping stage.
- Optionally there can be a combine/sort stage where all pairs with the same key value are grouped together.
- Then do a global reduction on the keys, yielding one or more of the corresponding values. This is the reduction stage.

We will now give a few examples of using MapReduce, and present the functional programming model that underlies the MapReduce abstraction.

2.11.3.1 Expressive power of the MapReduce model

The reduce part of the MapReduce model makes it a prime candidate for computing global statistics on a dataset. One example would be to count how many times each of a set of words appears in some set of documents. The function being mapped knows the set of words, and outputs for each document a pair of document name and a list with the occurrence counts of the words. The reduction then does a componentwise sum of the occurrence counts.

The combine stage of MapReduce makes it possible to transform data. An example is a 'Reverse Web-Link Graph': the map function outputs target-source pairs for each link to a target URL found in a page named "source". The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair target-list(source).

A less obvious example is computing PageRank (section 10.4) with MapReduce. Here we use the fact that the PageRank computation relies on a distributed sparse matrix-vector product. Each web page corresponds to a column of the web matrix W; given a probability p_j of being on page j, that page can then compute tuples $\langle i, w_{ij}p_j\rangle$. The combine stage of MapReduce then sums together $(Wp)_i = \sum_j w_{ij}p_j$.

Database operations can be implemented with MapReduce but since it has a relatively large latency, it is unlikely to be competitive with standalone databases, which are optimized for fast processing of a single query, rather than bulk statistics.

Sorting with MapReduce is considered in section 9.5.1.

For other applications see http://horicky.blogspot.com/2010/08/designing-algorithmis-for-map-reduce.html.

2.11.3.2 Mapreduce software

The implementation of MapReduce by Google was released under the name *Hadoop*. While it suited the Google model of single-stage reading and processing of data, it had considerable disadvantages for many other users:

- Hadoop would flush all its data back to disc after each MapReduce cycle, so for operations that take more than a single cycle the file system and bandwidth demands are too great.

- In computing center environments, where a user's data is not continuously online, the time required for loading data into *Hadoop File System (HDFS)* would likely overwhelm the actual analysis.

For these reasons, further projects such as Apache *Spark* (https://spark.apache.org/) offer caching of data.

2.11.3.3 Implementation issues

Implementing MapReduce on a distributed system has an interesting problem: the set of keys in the key-value pairs is dynamically determined. For instance, in the 'word count' type of applications above we do not *a priori* know the set of words. Therefore it is not clear which reducer process to send the pair to.

We could for instance use a *hash function* to determine this. Since every process uses the same function, there is not disagreement. This leaves the problem that a process does not know how many messages with key-value pairs to receive. The solution to this was described in section 7.5.6.

2.11.3.4 Functional programming

The mapping and reduction operations are readily implemented on any type of parallel architecture, using a combination of threading and message passing. However, at Google where this model was developed traditional parallelism was not attractive for two reasons. First of all, processors could fail during the computation, so a traditional model of parallelism would have to be enhanced with *fault tolerance* mechanisms. Secondly, the computing hardware could already have a load, so parts of the computation may need to be migrated, and in general any type of synchronization between tasks would be very hard.

MapReduce is one way to abstract from such details of parallel computing, namely through adopting a functional programming model. In such a model the only operation is the evaluation of a function, applied to some arguments, where the arguments are themselves the result of a function application, and the result of the computation is again used as argument for another function application. In particular, in a strict functional model there are no variables, so there is no static data.

A function application, written in *Lisp* style as `(f a b)` (meaning that the function `f` is applied to arguments `a` and `b`) would then be executed by collecting the inputs from whereven they are to the processor that evaluates the function `f`. The mapping stage of a MapReduce process is denoted

```
(map f (some list of arguments))
```

and the result is a list of the function results of applying `f` to the input list. All details of parallelism and of guaranteeing that the computation successfully finishes are handled by the `map` function.

Now we are only missing the reduction stage, which is just as simple:

```
(reduce g (map f (the list of inputs)))
```

The `reduce` function takes a list of inputs and performs a reduction on it.

The attractiveness of this functional model lies in the fact that functions can not have *side effects*: because they can only yield a single output result, they can not change their environment, and hence there is no coordination problem of multiple tasks accessing the same data.

Thus, MapReduce is a useful abstraction for programmers dealing with large amounts of data. Of course, on an implementation level the MapReduce software uses familiar concepts such as decomposing the data space, keeping a work list, assigning tasks to processors, retrying failed operations, et cetera.

2.11.4 The top500 list

There are several informal ways of measuring just 'how big' a computer is. The most popular is the TOP500 list, maintained at http://www.top500.org/, which records a computer's performance on the *Linpack benchmark* [51]. *Linpack* is a package for linear algebra operations, no longer in common use, since it has been superseded by *Lapack* for shared memory and *Scalapack* for distributed memory computers. The benchmark operation is the solution of a (square, nonsingular, dense) linear system through LU factorization with partial pivoting, with subsequent forward and backward solution.

The LU factorization operation is one that has great opportunity for cache reuse, since it is based on the matrix-matrix multiplication kernel discussed in section 1.6.1.2. (Subsequently using the factorization to solve a linear system involves recursion and is therefore harder to optimize; however, it is of a lower order of complexity, so of lesser importance, and consequently not discussed in this book.) It also has the property that the amount of work outweighs the amount of memory traffic and network communication: $O(n^3)$ versus $O(n^2)$. As a result, the Linpack benchmark is likely to run at a substantial fraction of the peak speed of the machine. Another way of phrasing this is to say that the Linpack benchmark is a *CPU-bound* or *compute-bound* algorithm.

Typical efficiency figures are between 60 and 90 percent. However, it should be noted that many scientific codes do not feature the dense linear solution kernel, so the performance on this benchmark is not indicative of the performance on a typical code. Linear system solution through iterative methods (section 5.5), for instance, is much less efficient in a flops-per-second sense, being dominated by the bandwidth between CPU and memory.

One implementation of the Linpack benchmark that is often used is 'High-Performance LINPACK' (http://www.netlib.org/benchmark/hpl/), which has several parameters such as blocksize that can be chosen to tune the performance.

2.11.4.1 *The top500 list as a recent history of supercomputing*

The top500 list offers a history of almost 20 years of supercomputing. In this section we will take a brief look at historical developments[13]. First of all, figure 2.43 shows the evolution of architecture types by charting what portion of the aggregate peak performance of the whole list is due to each type.

- Vector machines feature a relatively small number of very powerful vector pipeline processors (section 2.3.1.1). This type of architecture has largely disappeared; the last major machine of this type was the Japanese *Earth Simulator* which is seen as the spike in the graph around 2002, and which was at the top of the list for two years.
- Micro-processor based architectures get their power from the large number of processors in one machine. The graph distinguishes between *x86* (*Intel* and *AMD* processors with the exception of the *Intel Itanium*) processors and others; see also the next graph.

13. The graphs contain John McCalpin's analysis of the top500 data.

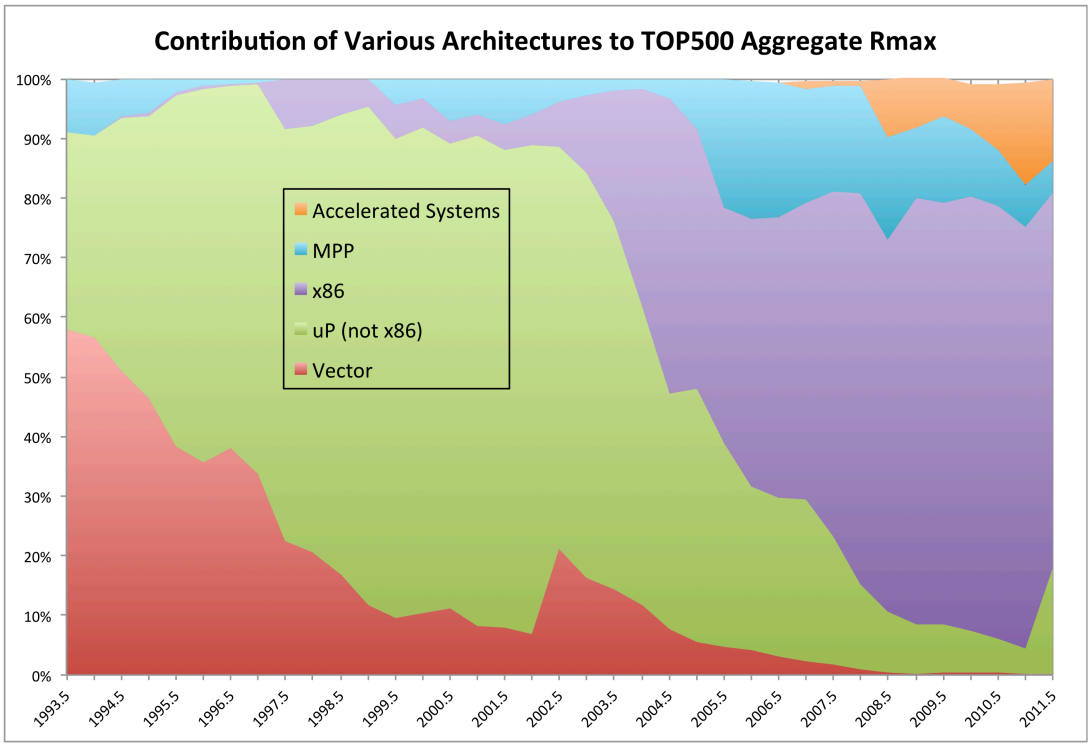


Figure 2.43: Evolution of the architecture types on the top500 list.

- A number of systems were designed as highly scalable architectures: these are denoted MPP for 'massively parallel processor'. In the early part of the timeline this includes architectures such as the *Connection Machine*, later it is almost exclusively the *IBM BlueGene*.
- In recent years 'accelerated systems' are the upcoming trend. Here, a processing unit such as a *GPU* is attached to the networked main processor.

Next, figure 2.44 shows the dominance of the *x86* processor type relative to other micro-processors. (Since we classified the *IBM BlueGene* as an MPP, its processors are not in the 'Power' category here.)

Finally, figure 2.45 shows the gradual increase in core count. Here we can make the following observations:

- In the 1990s many processors consisted of more than one chip. In the rest of the graph, we count the number of cores per 'package', that is, per *socket*. In some cases a socket can actually contain two separate dies.
- With the advent of multi-core processors, it is remarkable how close to vertical the section in the graph are. This means that new processor types are very quickly adopted, and the lower core counts equally quickly completely disappear.
- For accelerated systems (mostly systems with GPUs) the concept of 'core count' is harder to define; the graph merely shows the increasing importance of this type of architecture.

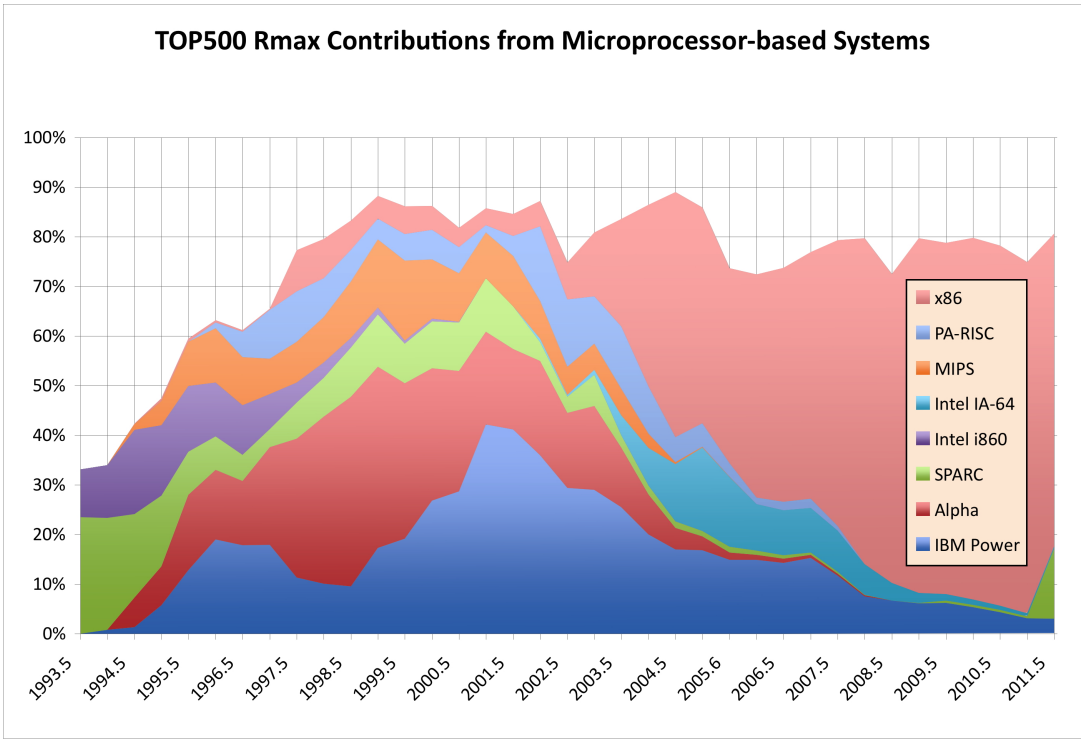


Figure 2.44: Evolution of the architecture types on the top500 list.

2.11.5 Heterogeneous computing

You have now seen several computing models: single core, shared memory multicore, distributed memory clusters, GPUs. These models all have in common that, if there is more than one instruction stream active, all streams are interchangeable. With regard to GPUs we need to refine this statement: all instruction stream *on the GPU* are interchangeable. However, a GPU is not a standalone device, but can be considered a *co-processor* to a *host processor*.

If we want to let the host perform useful work while the co-processor is active, we now have two different instruction streams or types of streams. This situation is known as *heterogeneous computing*. In the GPU case, these instruction streams are even programmed by a slightly different mechanisms – using *CUDA* for the GPU – but this need not be the case: the Intel Many Integrated Cores (MIC) architecture is programmed in ordinary C.

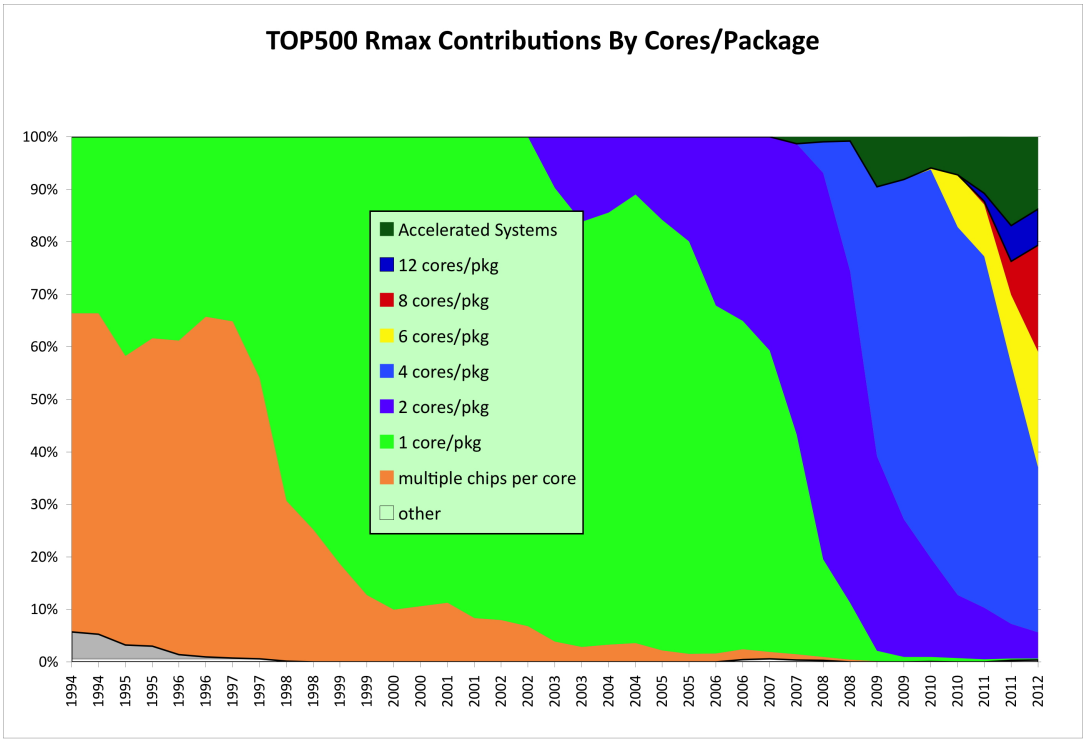


Figure 2.45: Evolution of the architecture types on the top500 list.

Chapter 3

Computer Arithmetic

Of the various types of data that one normally encounters, the ones we are concerned with in the context of scientific computing are the numerical types: integers (or whole numbers) $\ldots, -2, -1, 0, 1, 2, \ldots$, real numbers $0, 1, -1.5, 2/3, \sqrt{2}, \log 10, \ldots$, and complex numbers $1 + 2i, \sqrt{3} - \sqrt{5}i, \ldots$. Computer hardware is organized to give only a certain amount of space to represent each number, in multiples of bytes, each containing 8 bits. Typical values are 4 bytes for an integer, 4 or 8 bytes for a real number, and 8 or 16 bytes for a complex number.

Since only a certain amount of memory is available to store a number, it is clear that not all numbers of a certain type can be stored. For instance, for integers only a range is stored. (Languages such as *Python* have arbitrarily *large integers*, but this has no hardware support.) In the case of real numbers, even storing a range is not possible since any interval $[a, b]$ contains infinitely many numbers. Therefore, any *representation of real numbers* will cause gaps between the numbers that are stored. Calculations in a computer are sometimes described as *finite precision arithmetic*, to reflect that in general you not store numbers with infinite precision.

Since many results are not representable, any computation that results in such a number will have to be dealt with by issuing an error or by approximating the result. In this chapter we will look at how finite precision is realized in a computer, and the ramifications of such approximations of the 'true' outcome of numerical calculations.

For detailed discussions,

- the ultimate reference to floating point arithmetic is the *IEEE 754* standard [111];
- for secondary reading, see the book by Overton [158]; it is easy to find online copies of the essay by Goldberg [78];
- for extensive discussions of round-off error analysis in algorithms, see the books by Higham [104] and Wilkinson [196].

3.1 Bits

At the lowest level, computer storage and computer numbers are organized in *bit*s. A bit, short for 'binary digit', can have the values zero and one. Using bits we can then express numbers in *binary* notation:

$$10010_2 \equiv 18_{10} \tag{3.1}$$

where the subscript indicates the base of the number system, and in both cases the rightmost digit is the least significant one.

The next organizational level of memory is the *byte*: a byte consists of eight bits, and can therefore represent the values $0 \cdots 255$.

Exercise 3.1. Use bit operations to test whether a number is odd or even.
Can you think of more than one way?

3.2 Integers

In scientific computing, most operations are on real numbers. Computations on integers rarely add up to any serious computation load, except in applications such as cryptography. There are also applications such as 'particle-in-cell' that can be implemented with bit operations. However, integers are still encountered in indexing calculations.

Integers are commonly stored in 16, 32, or 64 bits, with 16 becoming less common and 64 becoming more and more so. The main reason for this increase is not the changing nature of computations, but the fact that integers are used to index arrays. As the size of data sets grows (in particular in parallel computations), larger indices are needed. For instance, in 32 bits one can store the numbers zero through $2^{32}-1 \approx 4 \cdot 10^9$. In other words, a 32 bit index can address 4 gigabytes of memory. Until recently this was enough for most purposes; these days the need for larger data sets has made 64 bit indexing necessary.

When we are indexing an array, only positive integers are needed. In general integer computations, of course, we need to accommodate the negative integers too. We will now discuss several strategies for implementing negative integers. Our motivation here will be that arithmetic on positive and negative integers should be as simple as on positive integers only: the circuitry that we have for comparing and operating on bitstrings should be usable for (signed) integers.

There are several ways of implementing negative integers. The simplest solution is to reserve one bit as a *sign bit*, and use the remaining 31 (or 15 or 63; from now on we will consider 32 bits the standard) bits to store the absolute magnitude. By comparison, we will call the straightforward interpretation of bitstrings as *unsigned* integers.

bitstring	$00\cdots0$	...	$01\cdots1$	$10\cdots0$	...	$11\cdots1$
interpretation as unsigned int	0	...	$2^{31}-1$	2^{31}	...	$2^{32}-1$
interpretation as signed integer	0	$\cdots$	$2^{31}-1$	-0	$\cdots$	$-(2^{31}-1)$

(3.2)

This scheme has some disadvantages, one being that there is both a positive and negative number zero. This means that a test for equality becomes more complicated than simply testing for equality as a bitstring. More importantly, in the second half of the bitstring, the interpretation as signed integer decreases, going to the right. This means that a test for greater-than becomes complex; also adding a positive number to a negative number now has to be treated differently from adding it to a positive number.

Another solution would be to let an unsigned number n be interpreted as $n - B$ where B is some plausible bias, for instance 2^{31}.

bitstring	$00\cdots0$	...	$01\cdots1$	$10\cdots0$	...	$11\cdots1$
interpretation as unsigned int	0	...	$2^{31}-1$	2^{31}	...	$2^{32}-1$
interpretation as shifted int	-2^{31}	...	-1	0	...	$2^{31}-1$

(3.3)

This shifted scheme does not suffer from the ±0 problem, and numbers are consistently ordered. However, if we compute $n - n$ by operating on the bitstring that represents n, we do not get the bitstring for zero.

To ensure this desired behavior, we instead rotate the number line with positive and negative numbers to put the pattern for zero back at zero. The resulting scheme, which is the one that is used most commonly, is called *2's complement*. Using this scheme, the representation of integers is formally defined as follows.

Definition 2 *Let n be an integer, then the 2's complement 'bit pattern' $\beta(n)$ is a non-negative integer defined as follows:*

- *If $0 \leq n \leq 2^{31}-1$, the normal bit pattern for n is used, that is*

$$0 \leq n \leq 2^{31}-1 \Rightarrow \beta(n) = n. \tag{3.4}$$

- *For $-2^{31} \leq n \leq -1$, n is represented by the bit pattern for $2^{32} - |n|$:*

$$-2^{31} \leq n \leq -1 \Rightarrow \beta(n) = 2^{32} - |n|. \tag{3.5}$$

We denote the inverse function that takes a bit pattern and interprets as integer with $\eta = \beta^{-1}$.

The following diagram shows the correspondence between bitstrings and their interpretation as 2's complement integer:

bitstring n	$00\cdots0$	...	$01\cdots1$	$10\cdots0$	...	$11\cdots1$
interpretation as unsigned int	0	...	$2^{31}-1$	2^{31}	...	$2^{32}-1$
interpretation $\beta(n)$ as 2's comp. integer	0	$\cdots$	$2^{31}-1$	-2^{31}	$\cdots$	-1

(3.6)

Some observations:

- There is no overlap between the bit patterns for positive and negative integers, in particular, there is only one pattern for zero.
- The positive numbers have a leading bit zero, the negative numbers have the leading bit set. This makes the leading bit act a little like a sign bit; however, you can not make a number change sign by flipping this bit.
- If you have a positive number n, you get $-n$ by flipping all the bits, and adding 1. (What is the inverse operation?)

Exercise 3.2. For the 'naive' scheme and the 2's complement scheme for negative numbers, give pseudocode for the comparison test $m < n$, where m and n are integers. Be careful to distinguish between all cases of m, n positive, zero, or negative.

3.2.1 Integer overflow

Adding two numbers with the same sign, or multiplying two numbers of any sign, may lead to a result that is too large or too small to represent. This is called *overflow*; see section 3.3.4 for the corresponding floating point phenomenon. The following exercise lets you explore the behavior of an actual program.

Exercise 3.3. Investigate what happens when you perform such a calculation. What does your compiler say if you try to write down a non-representable number explicitly, for instance in an assignment statement?

If you program this in C, it is worth noting that while you probably get an outcome that is understandable, the behavior of overflow in the case of signed quantities is actually *undefined* under the C standard.

3.2.2 Addition in two's complement

Let us consider doing some simple arithmetic on 2's complement integers. We start by assuming that we have hardware that works on unsigned integers. The goal is to see that we can use this hardware to do calculations on signed integers, as represented in 2's complement.

We consider the computation of $m+n$, where m, n are representable numbers:

$$0 \leq |m|, |n| < 2^{31}. \tag{3.7}$$

We distinguish different cases.

- The easy case is $0 < m, n$. In that case we perform the normal addition, and as long as the result stays under 2^{31}, we get the right result. If the result is 2^{31} or more, we have integer overflow, and there is nothing to be done about that.

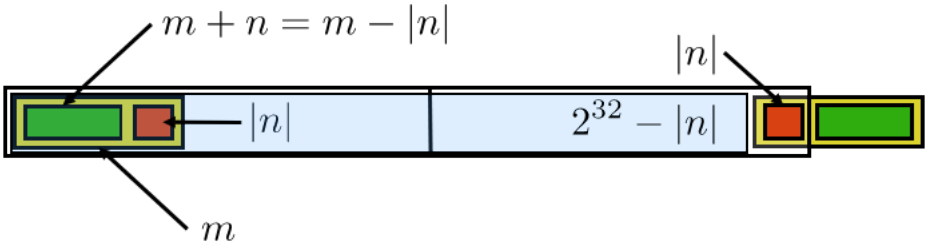


Figure 3.1: Addition with one positive and one negative number in 2's complement.

- Case $m > 0$, $n < 0$, and $m + n > 0$. Then $\beta(m) = m$ and $\beta(n) = 2^{32} - |n|$, so the unsigned addition becomes

$$\beta(m) + \beta(n) = m + (2^{32} - |n|) = 2^{32} + m - |n|. \tag{3.8}$$

  Since $m - |n| > 0$, this result is $> 2^{32}$. (See figure 3.1.) However, we observe that this is basically $m + n$ with the 33rd bit set. If we ignore this overflowing bit, we then have the correct result.
- Case $m > 0$, $n < 0$, but $m + n < 0$. Then

$$\beta(m) + \beta(n) = m + (2^{32} - |n|) = 2^{32} - (|n| - m). \tag{3.9}$$

  Since $|n| - m > 0$ we get

$$\eta\big(2^{32} - (|n| - m)\big) = -|(|n| - m)| = m - |n| = m + n. \tag{3.10}$$

3.2.3 Subtraction in two's complement

In exercise 3.2 above you explored comparing two integers. Let us now explore how subtracting numbers in two's complement is implemented. Consider $0 \leq m \leq 2^{31} - 1$ and $1 \leq n \leq 2^{31}$ and let us see what happens in the computation of $m - n$.

Suppose we have an algorithm for adding and subtracting unsigned 32-bit numbers. Can we use that to subtract two's complement integers? We start by observing that the integer subtraction $m - n$ becomes the unsigned addition $m + (2^{32} - n)$.

- Case: $m < |n|$. In this case, $m - n$ is negative and $1 \leq |m - n| \leq 2^{31}$, so the bit pattern for $m - n$ is

$$\beta(m - n) = 2^{32} - (n - m). \tag{3.11}$$

  Now, $2^{32} - (n - m) = m + (2^{32} - n)$, so we can compute $m - n$ in 2's complement by adding the bit patterns of m and $-n$ as unsigned integers:

$$\eta\big(\beta(m) + \beta(-n)\big) = \eta\big(m + (2^{32} - |n|)\big) = \eta\big(2^{32} + (m - |n|)\big) = \eta\left(2^{32} - \big|m - |n|\big|\right) = m - |n| = m + n. \tag{3.12}$$

- Case: $m > n$. Here we observe that $m + (2^{32} - n) = 2^{32} + m - n$. Since $m - n > 0$, this is a number $> 2^{32}$ and therefore not a legitimate representation of a negative number. However, if we store this number in 33 bits, we see that it is the correct result $m - n$, plus a single bit in the 33-rd position. Thus, by performing the unsigned addition, and ignoring the *overflow bit*, we again get the correct result.

In both cases we conclude that we can perform the subtraction $m - n$ by adding the unsigned number that represent m and $-n$ and ignoring overflow if it occurs.

3.2.4 Binary coded decimal

Decimal numbers are not relevant in scientific computing, but they are useful in financial calculations, where computations involving money absolutely have to be exact. Binary arithmetic is at a disadvantage here, since numbers such as $1/10$ are repeating fractions in binary. With a finite number of bits in the mantissa, this means that the number $1/10$ can not be represented exactly in binary. For this reason, *binary-coded-decimal* schemes were used in old IBM mainframes, and are in fact being standardized in revisions of *IEEE 754* [111, 110]; see also section 3.4.

In BCD schemes, one or more decimal digits are encoded in a number of bits. The simplest scheme would encode the digits $0 \ldots 9$ in four bits. This has the advantage that in a BCD number each digit is readily identified; it has the disadvantage that about $1/3$ of all bits are wasted, since 4 bits can encode the numbers $0 \ldots 15$. More efficient encodings would encode $0 \ldots 999$ in ten bits, which could in principle store the numbers $0 \ldots 1023$. While this is efficient in the sense that few bits are wasted, identifying individual digits in such a number takes some decoding. For this reason, BCD arithmetic needs hardware support from the processor, which is rarely found these days; one example is the IBM Power architecture, starting with the *IBM Power6*. Intel had 'BCD opcodes' that stored a decimal digit in 4 bits, so a byte could represent the numbers $0 \cdots 99$. These opcodes were removed in 64-bit mode.

3.3 Real numbers

In this section we will look at the principles of how real numbers are represented in a computer, and the limitations of various schemes. Section 3.4 will discuss the specific IEEE 754 solution, and section 3.5 will then explore the ramifications of this for arithmetic involving computer numbers.

3.3.1 They're not really real numbers

In the physical sciences, we usually work with real numbers, so it's convenient to pretend that computers can do this too. However, since numbers in a computer have only a finite number of bits, most real numbers can not be represented exactly. In fact, even many fractions can not be represented exactly, since they repeat; for instance, $1/3 = 0.333\ldots$, which is not representable in either decimal or binary. An illustration of this is given below in exercise 3.6.

Exercise 3.4. Some programming languages allow you to write loops with not just an integer, but also with a real number as 'counter'. Explain why this is a bad idea. Hint: when is the upper bound reached?

Whether a fraction repeats depends on the number system. (How would you write $1/3$ in ternary, or base 3, arithmetic?) In binary computers this means that fractions such as $1/10$, which in decimal arithmetic are terminating, are repeating. Since decimal arithmetic is important in financial calculations, some people care about the accuracy of this kind of arithmetic; see section 3.2.4 for a brief discussion of how this was realized in computer hardware.

Exercise 3.5. Show that each binary fraction, that is, a number of the form 0.01010111001_2, can exactly be represented as a terminating decimal fraction. What is the reason that not every decimal fraction can be represented as a binary fraction?

Exercise 3.6. Above, it was mentioned that many fractions are not representable in binary. Illustrate that by dividing a number first by 7, then multiplying it by 7 and dividing by 49. Try as inputs 3.5 and 3.6.

3.3.2 Representation of real numbers

Real numbers are stored using a scheme that is analogous to 'scientific notation', where a number is represented as a *significand* and an *exponent*, for instance $6.022 \cdot 10^{23}$, which has a significand 6022 with a *radix point* after the first digit, and an exponent 23. This number stands for

$$6.022 \cdot 10^{23} = \left[6 \times 10^0 + 0 \times 10^{-1} + 2 \times 10^{-2} + 2 \times 10^{-3}\right] \cdot 10^{23}. \tag{3.13}$$

For a general approach, we introduce a *base* β, which is a small integer number, 10 in the preceding example, and 2 in computer numbers. With this, we write numbers as a sum of t terms:

$$\begin{aligned} x &= \pm 1 \times \left[d_1\beta^0 + d_2\beta^{-1} + d_3\beta^{-2} + \cdots + d_t\beta^{-t+1}b\right] \times \beta^e \\ &= \pm\Sigma_{i=1}^{t} d_i\beta^{1-i} \times \beta^e \end{aligned} \tag{3.14}$$

where the components are

- the *sign bit*: a single bit storing whether the number is positive or negative;

- β is the base of the number system;
- $0 \leq d_i \leq \beta - 1$ the digits of the *mantissa* or *significand* – the location of the radix point (decimal point in decimal numbers) is implicitly assumed to be immediately following the first digit d_1;
- t is the length of the mantissa;
- $e \in [L, U]$ exponent; typically $L < 0 < U$ and $L \approx -U$.

Note that there is an explicit sign bit for the whole number, but the sign of the exponent is handled differently. For reasons of efficiency, e is not a signed number; instead it is considered as an unsigned number in *excess* of a certain minimum value. For instance, with 8 bits for the exponent we can represent the numbers $0 \cdots 256$, but with an excess of -127 this is interpreted as $-127 \cdots 128$. Additionally, the highest and lowest exponent values can be used to express special values such as infinity.

3.3.2.1 *Some examples*

Let us look at some specific examples of floating point representations. Base 10 is the most logical choice for human consumption, but computers are binary, so base 2 predominates there. Old IBM mainframes grouped bits to make for a base 16 representation.

	β	t	L	U
IEEE single precision (32 bit)	2	23	−126	127
IEEE double precision (64 bit)	2	53	−1022	1023
Old Cray 64 bit	2	48	−16383	16384
IBM mainframe 32 bit	16	6	−64	63
Packed decimal	10	50	−999	999

(3.15)

Of these, the single and double precision formats are by far the most common. We will discuss these in section 3.4 and further; for packed decimal, see section 3.2.4.

3.3.3 Normalized and unnormalized numbers

The general definition of floating point numbers, equation (3.14), leaves us with the problem that numbers can have more than one representation. For instance, $.5 \times 10^2 = .05 \times 10^3$. Since this would make computer arithmetic needlessly complicated, for instance in testing equality of numbers, we use *normalized floating point numbers*. A number is normalized if its first digit is nonzero. The implies that the mantissa part is $1 \leq x_m < \beta$; in binary $1 \leq x_m < 2$.

A practical implication in the case of binary numbers is that the first digit is always 1, so we do not need to store it explicitly. This means that every normalized floating point number is of the form

$$1.d_1 d_2 \ldots d_t \times 2^e \tag{3.16}$$

and only the digits $d_1 d_2 \ldots d_t$ are stored.

Exercise 3.7. With quantities t, e, what is the smallest positive normalized number?

Unnormalized numbers do exist, but only for numbers smaller than the smallest positive normalized number. See section 3.3.4.3.

3.3.4 Limitations: overflow and underflow

Since we use only a finite number of bits to store floating point numbers, not all numbers can be represented. The ones that can not be represented fall into two categories: those that are too large or too small (in some sense), and those that fall in the gaps.

The second category, where a computational result has to be rounded or truncated to be representable, is the basis of the field of round-off error analysis. We will study this in some detail below.

Numbers can be too large or too small in the following ways.

3.3.4.1 Overflow

The largest number we can store has every digit equal to β:

	unit	fractional			exponent
digit	$\beta-1$	$\beta-1$	$\cdots$	$\beta-1$	
value	1	β^{-1}	$\cdots$	$\beta^{-(t-1)}$	U

(3.17)

which adds up to

$$(\beta-1)\cdot 1+(\beta-1)\cdot\beta^{-1}+\cdots+(\beta-1)\cdot\beta^{-(t-1)}=\beta-\beta^{-(t-1)}, \tag{3.18}$$

and the smallest number (that is, the most negative) is $-\left(\beta-\beta^{-(t-1)}\right)$; anything larger than the former or smaller than the latter causes a condition called *overflow*.

3.3.4.2 Underflow

The number closest to zero is $\beta^{-(t-1)}\cdot\beta^{L}$.

	unit	fractional			exponent
digit	0	0	$\cdots$	$0,\beta-1$	$\emptyset$
value	0	0	$\cdots$	$0,\beta^{-(t-1)}$	L

(3.19)

A computation that has a result less than that (in absolute value) causes a condition called *underflow*.

The above 'smallest' number can not actually exist if we work with normalized numbers. Therefore, we also need to look at 'gradual underflow'.

3.3.4.3 Gradual underflow

The normalized number closest to zero is $1\cdot\beta^{L}$.

	unit	fractional			exponent
digit	1	0	⋯	0	∅
value	1	0	⋯	0	L

(3.20)

Trying to compute a number less than that in absolute value is sometimes handled by using *subnormal* (or *denormalized* or *unnormalized*) numbers, a process known as *gradual underflow*. In this case, a special value of the exponent indicates that the number is no longer normalized. In the case of IEEE standard arithmetic (see figure 3.3 for single precision) this is done through a zero exponent field.

However, this is typically tens or hundreds of times slower than computing with regular floating point numbers[1] . Not all processors have hardware support for gradual underflow. Ones that do include the *IBM Power6* (and up), *NVidia Fermi* (and up), but processors can have their default set to apply *flush-to-zero*. Section 6.3.1 explores performance implications.

3.3.4.4 What happens with over/underflow?

The occurrence of overflow or underflow means that your computation will be 'wrong' from that point on. Underflow will make the computation proceed with a zero where there should have been a nonzero; overflow is represented as `Inf`, short for 'infinite'.

Exercise 3.8. For real numbers x, y, the quantity $g = \sqrt{(x^2 + y^2)/2}$ satisfies

$$g \leq \max\{|x|, |y|\} \tag{3.21}$$

so it is representable if x and y are. What can go wrong if you compute g using the above formula? Can you think of a better way?

Computing with `Inf` is possible to an extent: adding two of those quantities will again give `Inf`. However, subtracting them gives `NaN`: 'not a number'. (See section 3.4.2.1.)

In none of these cases will the computation end: the processor will continue. However, you can query whether such an exception has occurred; see section 3.7.4.

3.3.4.5 Gradual underflow

Another implication of the normalization scheme is that we have to amend the definition of underflow (see section 3.3.4 above): any number less than $1 \cdot \beta^L$ now causes underflow.

3.3.5 Representation error

Let us consider a real number that is not representable in a computer's number system.

An unrepresentable number is approximated either by ordinary *rounding*, rounding up or down, or *truncation*. This means that a machine number $\tilde{x}$ is the representation for all x in an interval around it. With t

1. In real-time applications such as audio processing this phenomenon is especially noticeable; see http://phonophunk.com/articles/pentium4-denormalization.php?pg=3.

digits in the mantissa, this is the interval of numbers that differ from $\bar{x}$ in the $t+1$st digit. For the mantissa part we get:

$$\begin{cases} x \in \left[\tilde{x}, \tilde{x} + \beta^{-t+1}\right) & \text{truncation} \\ x \in \left[\tilde{x} - \frac{1}{2}\beta^{-t+1}, \tilde{x} + \frac{1}{2}\beta^{-t+1}\right) & \text{rounding} \end{cases} \tag{3.22}$$

If x is a number and $\tilde{x}$ its representation in the computer, we call $x - \tilde{x}$ the *representation error* or *absolute representation error*, and $\frac{x-\tilde{x}}{x}$ the *relative representation error*.

Often we are not interested in the sign of the error, so we may apply the terms error and relative error to $|x - \tilde{x}|$ and $|\frac{x-\tilde{x}}{x}|$ respectively.

Often we are only interested in bounds on the error. If ϵ is a bound on the error, we will write

$$\tilde{x} = x \pm \epsilon \underset{\text{D}}{\equiv} |x - \tilde{x}| \leq \epsilon \Leftrightarrow \tilde{x} \in [x - \epsilon, x + \epsilon] \tag{3.23}$$

For the relative error we note that

$$\tilde{x} = x(1 + \epsilon) \Leftrightarrow \left|\frac{\tilde{x} - x}{x}\right| \leq \epsilon \tag{3.24}$$

Let us consider an example in decimal arithmetic, that is, $\beta = 10$, and with a 3-digit mantissa: $t = 3$. The number $x = 1.256$ has a representation that depends on whether we round or truncate: $\tilde{x}_{\text{round}} = 1.26$, $\tilde{x}_{\text{truncate}} = 1.25$. The error is in the 4th digit: if $\epsilon = x - \tilde{x}$ then $|\epsilon| < \beta^{-(t-1)}$.

Exercise 3.9. The number in this example had no exponent part. What are the error and relative error if there had been one?

Exercise 3.10. In binary arithmetic the unit digit is always 1 as remarked above. How does that change the representation error?

3.3.6 Machine precision

Often we are only interested in the order of magnitude of the representation error, and we will write $\tilde{x} = x(1 + \epsilon)$, where $|\epsilon| \leq \beta^{-t}$. This maximum relative representation error is called the *machine precision*, or sometimes *machine epsilon*. Typical values are:

$$\begin{cases} \epsilon \approx 10^{-7} & \text{32-bit single precision} \\ \epsilon \approx 10^{-16} & \text{64-bit double precision} \end{cases} \tag{3.25}$$

(After you have studied the section on the IEEE 754 standard, can you derive these values from the number of bits in the mantissa?)

Machine precision can be defined another way: ϵ is the smallest number that can be added to 1 so that $1 + \epsilon$ has a different representation than 1. A small example shows how aligning exponents can shift a too small operand so that it is effectively ignored in the addition operation:

$$\begin{array}{rll} & 1.0000 & \times 10^0 \\ + & 1.0000 & \times 10^{-5} \\ \hline \end{array} \quad \Rightarrow \quad \begin{array}{rll} & 1.0000 & \times 10^0 \\ + & 0.00001 & \times 10^0 \\ \hline = & 1.0000 & \times 10^0 \end{array} \tag{3.26}$$

Yet another way of looking at this is to observe that, in the addition $x + y$, if the ratio of x and y is too large, the result will be identical to x.

The machine precision is the maximum attainable accuracy of computations: it does not make sense to ask for more than 6-or-so digits accuracy in single precision, or 15 in double.

Exercise 3.11. Write a small program that computes the machine epsilon. Does it make any difference if you set the *compiler optimization levels* low or high? (If you speak C++, can you solve this exercise with a single templated code?)

Exercise 3.12. The number $e \approx 2.72$, the base for the natural logarithm, has various definitions. One of them is

$$e = \lim_{n\to\infty} (1 + 1/n)^n. \tag{3.27}$$

Write a single precision program that tries to compute e in this manner. (Do not use the `pow` function: code the power explicitly.) Evaluate the expression for an upper bound $n = 10^k$ for some k. (How far do you let k range?) Explain the output for large n. Comment on the behavior of the error.

Exercise 3.13. The exponential function e^x can be computed as

$$e = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \tag{3.28}$$

Code this and try it for some positive x, for instance $x = 1, 3, 10, 50$. How many terms do you compute?
Now compute e^{-x} and from it e^x. Use for instance the same x values and number of iterations as for e^x.
What do you observe about the e^x versus e^{-x} computation? Explain.

3.4 The IEEE 754 standard for floating point numbers

Some decades ago, issues such as the length of the mantissa and the rounding behavior of operations could differ between computer manufacturers, and even between models from one manufacturer. This was obviously a bad situation from a point of portability of codes and reproducibility of results. The *IEEE 754* standard [111] codified all this in 1985, for instance stipulating 24 and 53 bits (before normalization) for the mantissa in single and double precision arithmetic, using a storage sequence of sign bit, exponent, mantissa; see figure 3.2.

Remark *The full name of the 754 standard is 'IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)'. It is also identical to IEC 559: 'Binary floating-point arithmetic for microprocessor systems', superseded by ISO/IEC/IEEE 60559:2011. IEEE 754 is a standard for binary arithmetic; there is a further standard,* IEEE 854*, that allows decimal arithmetic.*

Remark *'It was remarkable that so many hardware people there, knowing how difficult p754 would be, agreed that it should benefit the community at large. If it encouraged the production of floating-point software and*

sign	exponent	mantissa
p	$e = e_1 \cdots e_8$	$s = s_1 \ldots s_{23}$
31	$30 \cdots 23$	$22 \cdots 0$
$\pm$	2^{e-127} (except $e = 0, 255$)	$2^{-s_1} + \cdots + 2^{-s_{23}}$

sign	exponent	mantissa
s	$e_1 \cdots e_{11}$	$s_1 \ldots s_{52}$
63	$62 \cdots 52$	$51 \cdots 0$
$\pm$	2^{e-1023} (except $e = 0, 2047$)	$2^{-s_1} + \cdots + 2^{-s_{52}}$

(3.29)

Figure 3.2: Single and double precision definition.

eased the development of reliable software, it would help create a larger market for everyone's hardware. This degree of altruism was so astonishing that MATLAB's creator Dr. Cleve Moler used to advise foreign visitors not to miss the country's two most awesome spectacles: the Grand Canyon, and meetings of IEEE p754.' W. Kahan, `http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`.

The standard also declared the rounding behavior to be *correct rounding*: the result of an operation should be the rounded version of the exact result. See section 3.5.1.

3.4.1 Definition of the floating point formats

An inventory of the meaning of all bit patterns in *IEEE 754 single precision* is given in figure 3.3. Recall from section 3.3.3 above that for normalized numbers the first nonzero digit is a 1, which is not stored, so the bit pattern $d_1 d_2 \ldots d_t$ is interpreted as $1.d_1 d_2 \ldots d_t$.

The highest exponent is used to accommodate `Inf` and `Nan`; see section 3.4.2.1. The lowest exponent is used to accomodate unnormalized numbers; see section 3.3.4.3.

Exercise 3.14. Every programmer, at some point in their life, makes the mistake of storing a real number in an integer or the other way around. This can happen for instance if you call a function differently from how it was defined.

```
void a(float x) {....}
int main() {
  int i;
  .... a(i) ....
}
```

What happens when you print x in the function? Consider the bit pattern for a small integer, and use the table in figure 3.3 to interpret it as a floating point number. Explain that it will be a subnormal number.
(This is one of those errors you won't forget after you make it. In the future, whenever you see a number on the order of 10^{-305} you'll recognize that you probably made this error.)

These days, almost all processors adhere to the IEEE 754 standard. Early generations of the *NVidia Tesla GPU*s were not standard-conforming in single precision. The justification for this was that single precision

exponent	numerical value	range
$(0\cdots0)=0$	$\pm0.s_1\cdots s_{23}\times 2^{-126}$	$s=0\cdots01\Rightarrow 2^{-23}\cdot 2^{-126}=2^{-149}\approx 10^{-50}$
		...
		$s=1\cdots11\Rightarrow (1-2^{-23})\cdot 2^{-126}$
$(0\cdots01)=1$	$\pm1.s_1\cdots s_{23}\times 2^{-126}$	$s=0\cdots00\Rightarrow 1\cdot 2^{-126}\approx 10^{-37}$
$(0\cdots010)=2$	$\pm1.s_1\cdots s_{23}\times 2^{-125}$	
	...	
$(01111111)=127$	$\pm1.s_1\cdots s_{23}\times 2^{0}$	$s=0\cdots00\Rightarrow 1\cdot 2^{0}=1$
		...
		$s=0\cdots01\Rightarrow 1+2^{-23}\cdot 2^{0}=1+\epsilon$
		...
		$s=1\cdots11\Rightarrow (2-2^{-23})\cdot 2^{0}=2-\epsilon$
$(10000000)=128$	$\pm1.s_1\cdots s_{23}\times 2^{1}$	
	...	
$(11111110)=254$	$\pm1.s_1\cdots s_{23}\times 2^{127}$	
$(11111111)=255$	$s_1\cdots s_{23}=0\Rightarrow\pm\infty$	
	$s_1\cdots s_{23}\neq 0\Rightarrow NaN$	

(3.30)

Figure 3.3: Interpretation of single precision numbers depending on the exponent bit pattern.

is more likely used for graphics, where exact compliance matters less. For many scientific computations, double precision is necessary, since the precision of calculations gets worse with increasing problem size or runtime. This is true for Finite Difference (FD) calculations (chapter 4), but not for others such as the *Lattice Boltzmann Method (LBM)*.

3.4.2 Floating point exceptions

Various operations can give a result that is not representable as a floating point number. This situation is called an *exception*, and we say that an exception is *raised*. (Note: these are not C++ or python exceptions.) The result depends on the type of error, and the computation progresses normally. (It is possible to have the program be interrupted: section **??**.)

3.4.2.1 Not-a-Number

Apart from overflow and underflow, there are other exceptional situations. For instance, what result should be returned if the program asks for illegal operations such as $\sqrt{-4}$? The IEEE 754 standard has two special quantities for this: `Inf` for 'infinity' and `NaN` for 'not a number'. Whereas Infinity is the result of overflow or dividing by zero, not-a-number is the result of, for instance, subtracting infinity from

infinity.

To be precise, processors will represent as `NaN` ('not a number') the result of:

- Addition `Inf` $-$ `Inf`, but note that `Inf`+`Inf` gives `Inf`;
- Multiplication $0 \times$ `Inf`;
- Division $0/0$ or `Inf`$/$`Inf`;
- Remainder `Inf` *rem* x or x *rem* `Inf`;
- $\sqrt{x}$ for $x < 0$;
- Comparison $x < y$ or $x > y$ where any operand is `NaN`.

Since the processor can continue computing with such a number, it is referred to as a 'quiet `NaN`'. By contrast, some `NaN` quantities can cause the processor to generate an *interrupt* or *exception*. This is called a 'signaling `NaN`'.

If `NaN` appears in an expression, the whole expression will evaluate to that value. The rule for computing with `Inf` is a bit more complicated [78].

There are uses for a signaling NaN. You could for instance fill allocated memory with such a value, to indicate that it is uninitialized for the purposes of the computation. Any use of such a value is then a program error, and would cause an exception.

The *2008 revision* of IEEE 754 suggests using the most significant bit of a NaN as the `is_quiet` bit to distinguish between quiet and signaling NaNs.

See https://www.gnu.org/software/libc/manual/html_node/Infinity-and-NaN.html for treatment of `NaN` in the GNU compiler.

Remark *NaN and Inf do not always propagate:*

- *A finite number divided by Inf is zero;*
- *Various transcendental functions treat Inf correctly, such as* $\tan^{-1}(\text{Inf}) = \pi/2$.

The 2008 version of the standard treated the quiet NaN *as 'missing data', meaning that the minimum of 1 and a quiet* `NaN` *was 1. This led to min/max being non-associative. This was fixed in the 2019 standard where* `NaN`*s always propagate.*

3.4.2.2 Divide by zero

Division by zero results in `Inf`.

3.4.2.3 Overflow

This exception is raised if a result is not representable as a finite number.

3.4.2.4 Underflow

This exception is raised if a number is too small to be represented.

3.4.2.5 Inexact

This exception is raised for inexact results such as square roots, or overflow if that is not trapped.

3.5 Round-off error analysis

Numbers that are too large or too small to be represented, leading to overflow and underflow, are uncommon: usually computations can be arranged so that this situation will not occur. By contrast, the case that the result of a computation (even something as simple as a single addition) is not exactly representable is very common. Thus, looking at the implementation of an algorithm, we need to analyze the effect of such small errors propagating through the computation. This is commonly called *round-off error analysis*.

3.5.1 Correct rounding

The IEEE 754 standard, mentioned in section 3.4, does not only declare the way a floating point number is stored, it also gives a standard for the accuracy of operations such as addition, subtraction, multiplication, division. The model for arithmetic in the standard is that of *correct rounding*: the result of an operation should be as if the following procedure is followed:

- The exact result of the operation is computed, whether this is representable or not;
- This result is then rounded to the nearest computer number.

In short: the representation of the result of an operation is the rounded exact result of that operation. Of course, after two operations it no longer needs to hold that the computed result is the exact rounded version of the exact result.

If this statement sounds trivial or self-evident, consider subtraction as an example. In a decimal number system with two digits in the mantissa we could have a computation:

$$\begin{aligned} 1.0 - 9.4 \cdot 10^{-1} &= \\ 1.0 - 0.94 &= 0.06 \\ &= 0.6 \cdot 10^{-2} \end{aligned} \tag{3.31}$$

Note that in an intermediate step the mantissa .094 appears, which has one more digit than the two we declared for our number system. The extra digit is called a *guard digit*. Without a guard digit, this operation would have proceeded as $1.0 - 9.4 \cdot 10^{-1}$, where $9.4 \cdot 10^{-1}$ would be normalized to 0.9, giving a final result of 0.1, which is almost double the correct result.

Exercise 3.15. Consider the computation $1.0 - 9.5 \cdot 10^{-1}$, and assume again that numbers are rounded to fit the 2-digit mantissa. Why is this computation in a way a lot worse than the example?

One guard digit is not enough to guarantee correct rounding. An analysis that we will not reproduce here shows that three extra bits are needed [77].

Remark *The* 2008 revision *of the 754 standard declares that mathematical library functions should also be correctly rounded. However, not all compilers obey this.*

3.5.1.1 Mul-Add operations

In 2008, the IEEE 754 standard was revised to include the behavior of *Fused Multiply-Add (FMA)* operations, that is, operations of the form

$$c \leftarrow a * b + c. \tag{3.32}$$

This operation has a twofold motivation.

First, the FMA is potentially more accurate than a separate multiplication and addition, since it can use higher precision for the intermediate results, for instance by using the 80-bit *extended precision* format; section 3.8.3.

The standard here defines correct rounding to be that the result of this combined computation should be the rounded correct result. A naive implementation of this operations would involve two roundings: one after the multiplication and one after the addition[2].

Exercise 3.16. Can you come up with an example where correct rounding of an FMA is considerably more accurate than rounding the multiplication and addition separately? Hint: let the c term be of opposite sign as `a*b`, and try to force cancellation in the subtraction.

Secondly, FMA instructions are a way of getting higher performance: through pipelining we asymptotically get two operations per cycle. An FMA unit is then cheaper to construct than separate addition and multiplication units. Fortunately, the FMA occurs frequently in practical calculations.

Exercise 3.17. Can you think of some linear algebra operations that features FMA operations?

See section 1.2.1.2 for historic use of FMA in processors.

3.5.2 Addition

Addition of two floating point numbers is done in a couple of steps.

1. First the exponents are aligned: the smaller of the two numbers is written to have the same exponent as the larger number.
2. Then the mantissas are added.
3. Finally, the result is adjusted so that it again is a normalized number.

As an example, consider $1.00+2.00\times10^{-2}$. Aligning the exponents, this becomes $1.00+0.02 = 1.02$, and this result requires no final adjustment. We note that this computation was exact, but the sum $1.00+2.55\times10^{-2}$ has the same result, and here the computation is clearly not exact: the exact result is 1.0255, which is not representable with three digits to the mantissa.

In the example $6.15\times10^{1}+3.98\times10^{1} = 10.13\times10^{1} = 1.013\times10^{2} \rightarrow 1.01\times10^{2}$ we see that after addition of the mantissas an adjustment of the exponent is needed. The error again comes from truncating or rounding the first digit of the result that does not fit in the mantissa: if x is the true sum and $\tilde{x}$ the computed sum, then $\tilde{x} = x(1+\epsilon)$ where, with a 3-digit mantissa $|\epsilon| < 10^{-3}$.

Formally, let us consider the computation of $s = x_1+x_2$, and we assume that the numbers x_i are represented as $\tilde{x}_i = x_i(1+\epsilon_i)$. Then the sum s is represented as

$$\begin{aligned} \tilde{s} &= (\tilde{x}_1+\tilde{x}_2)(1+\epsilon_3) \\ &= x_1(1+\epsilon_1)(1+\epsilon_3) + x_2(1+\epsilon_2)(1+\epsilon_3) \\ &\approx x_1(1+\epsilon_1+\epsilon_3) + x_2(1+\epsilon_2+\epsilon_3) \\ &\approx s(1+2\epsilon) \end{aligned} \tag{3.33}$$

2. On the other hand, if the behavior of an application was 'certified' using a non-FMA architecture, the increased precision breaks the certification. Chip manufacturers have been known to get requests for a *double rounding* FMA to counteract this change in numerical behavior.

under the assumptions that all ϵ_i are small and of roughly equal size, and that both $x_i > 0$. We see that the relative errors are added under addition.

3.5.3 Multiplication

Floating point multiplication, like addition, involves several steps. In order to multiply two numbers $m_1 \times \beta^{e_1}$ and $m_2 \times \beta^{e_2}$, the following steps are needed.

- The exponents are added: $e \leftarrow e_1 + e_2$.
- The mantissas are multiplied: $m \leftarrow m_1 \times m_2$.
- The mantissa is normalized, and the exponent adjusted accordingly.

For example: $1.23 \cdot 10^0 \times 5.67 \cdot 10^1 = 0.69741 \cdot 10^1 \rightarrow 6.9741 \cdot 10^0 \rightarrow 6.97 \cdot 10^0$.

Exercise 3.18. Analyze the relative error of multiplication.

3.5.4 Subtraction

Subtraction behaves very differently from addition. Whereas in addition errors are added, giving only a gradual increase of overall roundoff error, subtraction has the potential for greatly increased error in a single operation.

For example, consider subtraction with 3 digits to the mantissa: $1.24 - 1.23 = 0.01 \rightarrow 1.00 \cdot 10^{-2}$. While the result is exact, it has only one significant digit[3]. To see this, consider the case where the first operand 1.24 is actually the rounded result of a computation that should have resulted in 1.235:

$$\begin{array}{lllll} .5 \times 2.47 - 1.23 & = 1.235 - 1.23 & = 0.005 & = 5.0 \cdot 10^{-3} & \text{exact} \\ & \downarrow & & & \\ & = 1.24 - 1.23 & = 0.01 & = 1. \cdot 10^{-2} & \text{computed} \end{array} \tag{3.34}$$

Now, there is a 100% error, even though the relative error of the inputs was as small as could be expected. Clearly, subsequent operations involving the result of this subtraction will also be inaccurate. We conclude that subtracting almost equal numbers is a likely cause of numerical roundoff.

There are some subtleties about this example. Subtraction of almost equal numbers is exact, and we have the correct rounding behavior of IEEE arithmetic. Still, the correctness of a single operation does not imply that a sequence of operations containing it will be accurate. While the addition example showed only modest decrease of numerical accuracy, the cancellation in this example can have disastrous effects. You'll see an example in section 3.6.1.

Exercise 3.19. Consider the iteration

$$x_{n+1} = f(x_n) = \begin{cases} 2x_n & \text{if } 2x_n < 1 \\ 2x_n - 1 & \text{if } 2x_n \ 1 \end{cases} \tag{3.35}$$

3. Normally, a number with 3 digits to the mantissa suggests an error corresponding to rounding or truncating the fourth digit. We say that such a number has 3 *significant digits*. In this case, the last two digits have no meaning, resulting from the normalization process.

Does this function have a fixed point, $x_0 \equiv f(x_0)$, or is there a cycle $x_1 = f(x_0)$, $x_0 \equiv x_2 = f(x_1)$ et cetera?
Now code this function. Is it possible to reproduce the fixed points? What happens with various starting points x_0. Can you explain this?

3.5.5 Associativity

The implementation of floating point number implies that simple arithmetic operation such as adding or multiplying no longer are *associative*, they way they are in mathematics. That is,

$$(a + b) + c \neq a + (b + c).$$

Let's consider a simple example, showing how associativity is affected by the rounding behavior of floating point numbers. Let floating point numbers be stored as a single digit for the mantissa, one digit for the exponent, and one guard digit; now consider the computation of $4 + 6 + 7$. Evaluation left-to-right gives:

$$\begin{aligned}
(4 \cdot 10^0 + 6 \cdot 10^0) + 7 \cdot 10^0 &\Rightarrow 10 \cdot 10^0 + 7 \cdot 10^0 && \text{addition} \\
&\Rightarrow 1 \cdot 10^1 + 7 \cdot 10^0 && \text{rounding} \\
&\Rightarrow 1.0 \cdot 10^1 + 0.7 \cdot 10^1 && \text{using guard digit} \\
&\Rightarrow 1.7 \cdot 10^1 \\
&\Rightarrow 2 \cdot 10^1 && \text{rounding}
\end{aligned} \tag{3.36}$$

On the other hand, evaluation right-to-left gives:

$$\begin{aligned}
4 \cdot 10^0 + (6 \cdot 10^0 + 7 \cdot 10^0) &\Rightarrow 4 \cdot 10^0 + 13 \cdot 10^0 && \text{addition} \\
&\Rightarrow 4 \cdot 10^0 + 1 \cdot 10^1 && \text{rounding} \\
&\Rightarrow 0.4 \cdot 10^1 + 1.0 \cdot 10^1 && \text{using guard digit} \\
&\Rightarrow 1.4 \cdot 10^1 \\
&\Rightarrow 1 \cdot 10^1 && \text{rounding}
\end{aligned} \tag{3.37}$$

The conclusion is that the sequence in which rounding and truncation is applied to intermediate results makes a difference.

Exercise 3.20. The above example used rounding. Can you come up with a similar example in an arithmetic system using truncation?

Usually, the evaluation order of expressions is determined by the definition of the programming language, or at least by the compiler. In section 3.6.5 we will see how in parallel computations the associativity is not so uniquely determined.

3.6 Examples of round-off error

From the above, the reader may got the impression that roundoff errors only lead to serious problems in exceptional circumstances. In this section we will discuss some very practical examples where the

inexactness of computer arithmetic becomes visible in the result of a computation. These will be fairly simple examples; more complicated examples exist that are outside the scope of this book, such as the instability of matrix inversion. The interested reader is referred to [104, 196].

3.6.1 Cancellation: the 'abc-formula'

As a practical example of cancellation, consider the quadratic equation $ax^2+bx+c = 0$ which has solutions $x = \frac{-b\pm\sqrt{b^2-4ac}}{2a}$. Suppose $b > 0$ and $b^2 \gg 4ac$ then $\sqrt{b^2-4ac} \approx b$ and the '+' solution will be inaccurate. In this case it is better to compute $x_- = \frac{-b-\sqrt{b^2-4ac}}{2a}$ and use $x_+ \cdot x_- = c/a$.

For an illustration, we consider the equation

$$f(x) = \epsilon x^2 - (1+\epsilon^2)x + \epsilon,$$

which has, for ϵ small, the roots

$$x_+ \approx \epsilon^{-1}, \qquad x_- \approx \epsilon.$$

We have two problems here. First of all, for ϵ less than machine precision we have $b \approx 1$, $b^2 - 4ac = 1$, and $x_- = 0$, which means that this solution is totally wrong. Even if the *discriminant* $b^2 - 4ac > 1$, we get problems with cancellation, and the function value is grossly inaccurate: for the textbook solution, $f(x_-) \approx \epsilon$, while for the solution without cancellation $f(x_-) \approx \epsilon^3$.

	textbook		accurate	
ϵ	x_-	$f(x_-)$	x_-	$f(x_-)$
10^{-3}	$1.000 \cdot 10^{-03}$	$-2.876 \cdot 10^{-14}$	$1.000 \cdot 10^{-03}$	$-2.168 \cdot 10^{-19}$
10^{-4}	$1.000 \cdot 10^{-04}$	$5.264 \cdot 10^{-14}$	$1.000 \cdot 10^{-04}$	0.000
10^{-5}	$1.000 \cdot 10^{-05}$	$-8.274 \cdot 10^{-13}$	$1.000 \cdot 10^{-05}$	$-1.694 \cdot 10^{-21}$
10^{-6}	$1.000 \cdot 10^{-06}$	$-3.339 \cdot 10^{-11}$	$1.000 \cdot 10^{-06}$	$-2.118 \cdot 10^{-22}$
10^{-7}	$9.992 \cdot 10^{-08}$	$7.993 \cdot 10^{-11}$	$1.000 \cdot 10^{-07}$	$1.323 \cdot 10^{-23}$
10^{-8}	$1.110 \cdot 10^{-08}$	$-1.102 \cdot 10^{-09}$	$1.000 \cdot 10^{-08}$	0.000
10^{-9}	0.000	$1.000 \cdot 10^{-09}$	$1.000 \cdot 10^{-09}$	$-2.068 \cdot 10^{-25}$
10^{-10}	0.000	$1.000 \cdot 10^{-10}$	$1.000 \cdot 10^{-10}$	0.000

(3.38)

Exercise 3.21. Write a program that computes the roots of the quadratic equation, both the 'textbook' way, and as described above.

- Let $b = -1$ and $a = -c$, and $4ac \downarrow 0$ by taking progressively smaller values for a and c.
- Print out the computed root, the root using the stable computation, and the value of $f(x) = ax^2 + bx + c$ in the computed root.

Now suppose that you don't care much about the actual value of the root: you want to make sure the residual $f(x)$ is small in the computed root. Let x^* be the exact root, then

$$f(x^* + h) \approx f(x^*) + hf'(x^*) = hf'(x^*). \tag{3.39}$$

Now investigate separately the cases $a \downarrow 0, c = -1$ and $a = -1, c \downarrow 0$. Can you explain the difference?

Exercise 3.22. Consider the functions

$$\begin{cases} f(x) = \sqrt{x+1} - \sqrt{x} \\ g(x) = 1/\left(\sqrt{x+1} + \sqrt{x}\right) \end{cases} \tag{3.40}$$

- Show that they are the same in exact arithmetic; however:
- Show that f can exhibit cancellation and that g has no such problem.
- Write code to show the difference between f and g. You may have to use large values for x.
- Analyze the cancellation in terms of x and machine precision. When are $\sqrt{x+1}$ and $\sqrt{x}$ less than ϵ apart? What happens then? (For a more refined analysis, when are they $\sqrt{\epsilon}$ apart, and how does that manifest itself?)
- The inverse function of $y = f(x)$ is

$$x = (y^2 - 1)^2/(4y^2) \tag{3.41}$$

  Add this to your code. Does this give any indication of the accuracy of the calculations?

Make sure to test your code in single and double precision. If you speak python, try the `bigfloat` package.

3.6.2 Summing series

The previous example was about preventing a large roundoff error in a single operation. This example shows that even gradual buildup of roundoff error can be handled in different ways.

Consider the sum $\sum_{n=1}^{10000} \frac{1}{n^2} = 1.644834\ldots$ and assume we are working with single precision, which on most computers means a machine precision of 10^{-7}. The problem with this example is that both the ratio between terms, and the ratio of terms to partial sums, is ever increasing. In section 3.3.6 we observed that a too large ratio can lead to one operand of an addition in effect being ignored.

If we sum the series in the sequence it is given, we observe that the first term is 1, so all partial sums ($\sum_{n=1}^{N}$ where $N < 10000$) are at least 1. This means that any term where $1/n^2 < 10^{-7}$ gets ignored since it is less than the machine precision. Specifically, the last 7000 terms are ignored, and the computed sum is 1.644725. The first 4 digits are correct.

However, if we evaluate the sum in reverse order we obtain the exact result in single precision. We are still adding small quantities to larger ones, but now the ratio will never be as bad as one-to-ϵ, so the smaller number is never ignored. To see this, consider the ratio of two terms subsequent terms:

$$\frac{n^2}{(n-1)^2} = \frac{n^2}{n^2 - 2n + 1} = \frac{1}{1 - 2/n + 1/n^2} \approx 1 + \frac{2}{n} \tag{3.42}$$

Since we only sum 10^5 terms and the machine precision is 10^{-7}, in the addition $1/n^2 + 1/(n-1)^2$ the second term will not be wholly ignored as it is when we sum from large to small.

Exercise 3.23. There is still a step missing in our reasoning. We have shown that in adding two subsequent terms, the smaller one is not ignored. However, during the calculation we add partial sums to the next term in the sequence. Show that this does not worsen the situation.

The lesson here is that series that are monotone (or close to monotone) should be summed from small to large, since the error is minimized if the quantities to be added are closer in magnitude. Note that this is the opposite strategy from the case of subtraction, where operations involving similar quantities lead to larger errors. This implies that if an application asks for adding and subtracting series of numbers, and we know a priori which terms are positive and negative, it may pay off to rearrange the algorithm accordingly.

Exercise 3.24. The sine function is defined as

$$\begin{aligned} \sin(x) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \\ &= \sum_{i\geq 0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}. \end{aligned} \tag{3.43}$$

Here are two code fragments that compute this sum (assuming that `x` and `nterms` are given):

```
double term = x, sum = term;
for (int i=1; i<=nterms; i+=2) {
  term *=
    - x*x / (double)((i+1)*(i+2));
  sum += term;
}
printf("Sum: %e\n\n",sum);
```

```
double term = x, sum = term;
double power = x, factorial = 1.,
    factor = 1.;
for (int i=1; i<=nterms; i+=2) {
  power *= -x*x;
  factorial *= (factor+1)*(factor+2);
  term = power / factorial;
  sum += term; factor += 2;
}
printf("Sum: %e\n\n",sum);
```

- Explain what happens if you compute a large number of terms for $x > 1$.
- Does either code make sense for a large number of terms?
- Is it possible to sum the terms starting at the smallest? Would that be a good idea?
- Can you come with other schemes to improve the computation of $\sin(x)$?

3.6.3 Unstable algorithms

We will now consider an example where we can give a direct argument that the algorithm can not cope with problems due to inexactly represented real numbers.

Consider the recurrence $y_n = \int_0^1 \frac{x^n}{x-5} dx = \frac{1}{n} - 5y_{n-1}$, which is monotonically decreasing; the first term can be computed as $y_0 = \ln 6 - \ln 5$.

Performing the computation in 3 decimal digits we get:

computation		correct result
$y_0 = \ln 6 - \ln 5 = .182\|322 \times 10^1 \ldots$		1.82
$y_1 = .900 \times 10^{-1}$		.884
$y_2 = .500 \times 10^{-1}$		.0580
$y_3 = .830 \times 10^{-1}$	going up?	.0431
$y_4 = -.165$	negative?	.0343

We see that the computed results are quickly not just inaccurate, but actually nonsensical. We can analyze why this is the case.

If we define the error ϵ_n in the n-th step as

$$\tilde{y}_n - y_n = \epsilon_n, \tag{3.44}$$

then

$$\tilde{y}_n = 1/n - 5\tilde{y}_{n-1} = 1/n + 5n_{n-1} + 5\epsilon_{n-1} = y_n + 5\epsilon_{n-1} \tag{3.45}$$

so $\epsilon_n \geq 5\epsilon_{n-1}$. The error made by this computation shows exponential growth.

3.6.4 Linear system solving

Sometimes we can make statements about the numerical precision of a problem even without specifying what algorithm we use. Suppose we want to solve a linear system, that is, we have an $n \times n$ matrix A and a vector b of size n, and we want to compute the vector x such that $Ax = b$. (We will actually considering algorithms for this in chapter 5.) Since the vector b will the result of some computation or measurement, we are actually dealing with a vector $\tilde{b}$, which is some perturbation of the ideal b:

$$\tilde{b} = b + \Delta b. \tag{3.46}$$

The perturbation vector Δb can be of the order of the machine precision if it only arises from representation error, or it can be larger, depending on the calculations that produced $\tilde{b}$.

We now ask what the relation is between the exact value of x, which we would have obtained from doing an exact calculation with A and b, which is clearly impossible, and the computed value $\tilde{x}$, which we get from computing with A and $\tilde{b}$. (In this discussion we will assume that A itself is exact, but this is a simplification.)

Writing $\tilde{x} = x + \Delta x$, the result of our computation is now

$$A\tilde{x} = \tilde{b} \tag{3.47}$$

or

$$A(x + \Delta x) = b + \Delta b. \tag{3.48}$$

Since $Ax = b$, we get $A\Delta x = \Delta b$. From this, we get (see appendix 14 for details)

$$\left\{ \begin{array}{ll} \Delta x & = A^{-1}\Delta b \\ Ax & = b \end{array} \right\} \Rightarrow \left\{ \begin{array}{ll} \|A\|\|x\| & \geq \|b\| \\ \|\Delta x\| & \leq \|A^{-1}\|\|\Delta b\| \end{array} \right. \Rightarrow \frac{\|\Delta x\|}{\|x\|} \leq \|A\|\|A^{-1}\| \frac{\|\Delta b\|}{\|b\|} \tag{3.49}$$

The quantity $\|A\|\|A^{-1}\|$ is called the *condition number* of a matrix. The bound (3.49) then says that any perturbation in the right hand side can lead to a perturbation in the solution that is at most larger by the condition number of the matrix A. Note that it does not say that the perturbation in x *needs* to be anywhere close to that size, but we can not rule it out, and in some cases it indeed happens that this bound is attained.

Suppose that b is exact up to machine precision, and the condition number of A is 10^4. The bound (3.49) is often interpreted as saying that the last 4 digits of x are unreliable, or that the computation 'loses 4 digits of accuracy'.

Equation (3.49) can also be interpreted as follows: when we solve a linear system $Ax = b$ we get an approximate solution $x + \Delta x$ which is the *exact* solution of a perturbed system $A(x + \Delta x) = b + \Delta b$. The fact that the perturbation in the solution can be related to the perturbation in the system, is expressed by saying that the algorithm exhibits *backwards stability*.

The analysis of the accuracy of linear algebra algorithms is a field of study in itself; see for instance the book by Higham [104].

3.6.5 Roundoff error in parallel computations

As we discussed in section 3.5.5, and as you saw in the above example of summing a series, addition in computer arithmetic is not *associative*. A similar fact holds for multiplication. The lack of associativity has an interesting consequence for parallel computations: the way a computation is spread over parallel processors influences the result.

As a very simple example, consider computing the sum of four numbers $a+b+c+d$. On a single processor execution proceeds left-to-right, corresponding to the following associativity:

$$((a + b) + c) + d. \tag{3.50}$$

On the other hand, spreading this computation over two processors, where processor 0 has a, b and processor 1 has c, d, corresponds to

$$((a + b) + (c + d)). \tag{3.51}$$

Generalizing this, we see that reduction operations will most likely give a different result on different numbers of processors. (The MPI standard declares that two program runs on the same set of processors should give the same result.) It is possible to circumvent this problem by replace a reduction operation by a *gather* operation to all processors, which subsequently do a local reduction. However, this increases the memory requirements for the processors.

There is an intriguing other solution to the parallel summing problem. If we use a mantissa of 4000 bits to store a floating point number, we do not need an exponent, and all calculations with numbers thus stored are exact since they are a form of fixed-point calculation [124, 125]. While doing a whole application with such numbers would be very wasteful, reserving this solution only for an occasional inner product calculation may be the solution to the reproducibility problem.

3.7 Computer arithmetic in programming languages

Different languages have different approaches to declaring integers and floating point numbers. Here we study some of the issues.

3.7.1 C/C++ data models

The C/C++ language has `short`, `int`, `float`, `double` types (see section 3.8.6 for complex), plus the `long` and `unsigned` modifiers on these. (Using `long` by itself is synonymous to `long int`.) The language standard gives no explicit definitions on these, but only the following restrictions:

- A short int is at least 16 bits;
- An integer is at least 16 bits, which was the case in the old days of the *DEC PDP-11*, but nowadays they are commonly 32 bits;
- A long integer is at least 32 bits, but often 64;
- A `long long` integer is at least 64 bits.
- If you need only one byte for your integer, you can use a `char`.

Additionally, pointers are related to unsigned integers; see section 3.7.2.3.

There are a number of conventions, called *data model*s, for the actual sizes of the various integer types.

- *LP32* (or '2/4/4') has 16-bit `int`s, and 32 bits for `long int` and pointers. This is used in the *Win16* API.
- *ILP32* (or '4/4/4') has 32 bits for `int`, `long`, and pointers. Only `long long` is 64 bits. This is used in the *Win32* API and on most 32-bit Unix or Unix-like systems.
- *LLP64* (or '4/4/8') has 32 bits for `int` and `long` (!!!); 64 bits are used for `long long`, `size_t` and pointers. This is used in the *Win64* API.
- *LP64* (or '4/8/8') has 32-bit `int`s, and 64 bits for `long`, `long long`, `size_t`, and pointers.
- *SILP64* (or '8/8/8') uses 64 bits for all of `short`, `int`, `long`, `long long`, `size_t`, and pointers. This only existed on some early Unix systems such as *Cray UNICOS*. The related *ILP64* used 16 bits for `short`.

3.7.2 C/C++

Certain type handling is common to the C and C++ languages; see below for mechanisms that are exclusive to C++.

3.7.2.1 Bits

The C logical operators and their bit variants are:

	boolean	bitwise
and	&&	&
or	||	|
not	!	
xor		^

The following *bit shift* operations are available:

left shift	<<
right shift	>>

You can do arithmetic with bit operations:

- Left-shift is multiplication by 2:

  ```
  i_times_2 = i<<1;
  ```

- Extract bits:

  ```
  i_mod_8 = i & 7
  ```

  (How does that last one work?)

Exercise 3.25. Bit shift operations are normally applied to unsigned quantities. Are there extra complications when you use bitshifts to multiply or divide by 2 in 2's-complement?

The following code fragment is useful for printing the bit pattern of numbers:

MISSING SNIPPET cprintbits in codesnippetsdir=snippets.code

Sample usage:

MISSING SNIPPET printbits5 in codesnippetsdir=snippets.code

3.7.2.2 *Printing bit patterns*

While C++ has a `hexfloat` format, this is not an intuitive way of displaying the bit pattern of a binary number. Here is a handy routine for displaying the actual bits.

Code:

```
// 754/bitprint.cxx
void format(const std::string  &s)
{
  // sign bit
  std::cout << s.substr(0,1)  << ' ';
  // exponent
  std::cout << s.substr(1,8);
  // mantissa in groups of 4
  for(int walk=9;walk<32;walk+=4)
    std::cout << ' ' << s.substr(walk,4);
  // newline
  std::cout << "\n";
}
    /* ... */
  uint32_t    u;
  std::memcpy(&u,&d,sizeof(u));
  std::bitset<32> b{u};
  std::stringstream s;
  s << std::hexfloat << b << '\n';
  format(s.str());
```

Output:

```
[code/754] bitprint:

Binary output of 3.14:

hexfloat:0x1.91eb86p+1
S eeeeeeee mmmm mmmm mmmm mmmm
    ↪mmmm mmm
0 10000000 1001 0001 1110 1011
    ↪1000 011
```

3.7.2.3 Integers and floating point numbers

The `sizeof()` operator gives the number of bytes used to store a datatype.

Floating point types are specified in `float.h`.

C integral types with specified storage exist: constants such as `int64_t` are defined by `typedef` in `stdint.h`.

The constant `NAN` is declared in `math.h`. For checking whether a value is `NaN`, use `isnan()`.

3.7.2.4 Changing rounding behavior

As mandated by the 754 standard, *rounding behavior* should be controllable. In C99, the API for this is contained in `fenv.h` (or for C++ `cfenv`):

```
#include <fenv.h>

int roundings[] =
  {FE_TONEAREST, FE_UPWARD, FE_DOWNWARD, FE_TOWARDZERO};
rchoice = ....
int status = fesetround(roundings[rchoice]);
```

Setting the rounding behavior can serve as a quick test for the stability of an algorithm: if the result changes appreciably between two different rounding strategies, the algorithm is likely not stable.

3.7.3 Limits

For C, the *numerical ranges* of C integers are defined in `limits.h`, typically giving an upper or lower bound. For instance, `INT_MAX` is defined to be 32767 or greater.

In C++ you can still use the C header `limits.h` or `climits`, but it's better to use `std::numeric_limits`, which is templated over the types. (See *Introduction to Scientific Programming book, section 24.2* for details.)

For instance

```
std::numerical_limits<int>.max();
```

3.7.4 Exceptions

The C/C++ languages allow access to floating point behavior through the `fenv.h` or `cfenv` header. This can be used to set rounding behavior, or to query *floating point exception*, such as overflow or divide by zero.

3.7.4.1 Testing

The header defines symbolic names for exceptions to be tested, such as `FE_DIVBYZERO`, `FE_OVERFLOW`, `FE_UNDERFLOW` . The actual testing is done through the function `fetestexcept`:

Code:

```
// fenv/divzero.cpp
float result = 1.f/zero;
cout << "result: " << result << '\n';
if (std::fetestexcept(FE_DIVBYZERO))
  std::cout << "division by zero reported\n";
else
  std::cout
    << "division by zero not reported\n";
```

Output:

```
[code/fenv] divzero:

result: inf
division by zero reported
```

3.7.4.2 *Changing behavior*

Another functionality offered by the header is to change rounding behavior, with values *FE_DOWNWARD*, *FE_TONEAREST*, *FE_TOWARDZERO*, *FE_UPWARD*.

Code:

```
// fenv/round.cpp
fesetround(FE_UPWARD);
float thirdup = one/3.f;
fesetround(FE_DOWNWARD);
float thirddn = one/3.f;
cout << "difference: "
     << thirdup-thirddn << '\n';
```

Output:

```
[code/fenv] round:

difference: 2.98023e-08
```

(Can you relate the output of this code to the value of machine epsilon?)

3.7.4.3 *Exceptions defined*

Exceptions defined:

- *FE_DIVBYZERO* pole error occurred in an earlier floating-point operation.
- *FE_INEXACT* inexact result: rounding was necessary to store the result of an earlier floating-point operation.
- *FE_INVALID* domain error occurred in an earlier floating-point operation.
- *FE_OVERFLOW* the result of the earlier floating-point operation was too large to be representable.
- *FE_UNDERFLOW* the result of the earlier floating-point operation was subnormal with a loss of precision.
- *FE_ALL_EXCEPT* bitwise OR of all supported floating-point exceptions .

Usage:

```
std::feclearexcept(FE_ALL_EXCEPT);
if(std::fetestexcept(FE_UNDERFLOW)) { /* ... */ }
```

In C++, `std::numeric_limits<double>::quiet_NaN()` is declared in the `limits` header, which is meaningful if `std::numeric_limits::has_quiet_NaN` is true, which is the case if `std::numeric_limits::is_iec559` is true. (ICE 559 is essentially IEEE 754; see section 3.4.)

The same module also has `infinity()` and `signaling_NaN()`.

For checking whether a value is `NaN`, use `std::isnan()` from `cmath` in C++.

See further http://en.cppreference.com/w/cpp/numeric/math/nan.

3.7.4.4 *Compiler-specific behavior*

Trapping exceptions can sometimes be specified by the compiler. For instance, the *gcc* compiler can *trap* exceptions by the flag `-ffpe-trap=list`; see https://gcc.gnu.org/onlinedocs/gfortran/Debugging-Options.html.

3.7.5 Compiler flags for fast math

Various compilers have an option for *fast math* optimizations.

- GCC and Clang: `-ffast-math`
- Intel: `-fp-model=fast` (default)
- MSVC: `/fp:fast`

This typically covers the following cases:

- Finite math: assume that `Inf` and `NaN` don't occur. This means that the test `x==x` is always true.
- Associative math: this allows rearranging the order of operations in an arithmetic expression. This is known as *re-association*, and is for instance beneficial for vectorization. However, as you saw in section 3.5.5, this can change the result of a computation. Also, it makes *compensated summation* impossible; section 3.8.1.
- Flushing subnormals to zero.

Extensive discussion: https://simonbyrne.github.io/notes/fastmath/

3.7.6 Fortran

3.7.6.1 *Variable 'kind's*

Fortran has several mechanisms for indicating the precision of a numerical type.

```
integer(2) :: i2
integer(4) :: i4
integer(8) :: i8

real(4) :: r4
real(8) :: r8
real(16) :: r16

complex(8) :: c8
complex(16) :: c16
complex*32 :: c32
```

This often corresponds to the number of bytes used, **but not always**. It is technically a numerical *kind selector*, and it is nothing more than an identifier for a specific type.

```
integer, parameter :: k9 = selected_real_kind(9)
real(kind=k9) :: r
r = 2._k9; print *, sqrt(r) ! prints 1.4142135623730
```

The 'kind' values will usually be 4,8,16 but this is compiler dependent.

3.7.6.2 *Rounding behavior*

In Fortran2003 the function `IEEE_SET_ROUNDING_MODE` is available in the `IEEE_ARITHMETIC` module.

3.7.6.3 *C99 and Fortran2003 interoperability*

Recent standards of the C and Fortran languages incorporate the C/Fortran interoperability standard, which can be used to declare a type in one language so that it is compatible with a certain type in the other language.

3.7.7 Round-off behavior in programming

From the above discussion it should be clear that some simple statements that hold for mathematical real numbers do not hold for floating-point numbers. For instance, in floating-point arithmetic

$$(a+b)+c \neq a+(b+c). \tag{3.52}$$

This implies that a compiler can not perform certain optimizations without it having an effect on round-off behavior[4] . In some codes such slight differences can be tolerated, for instance because the method has built-in safeguards. For instance, the stationary iterative methods of section 5.5 damp out any error that is introduced.

On the other hand, if the programmer has written code to account for round-off behavior, the compiler has no such liberties. This was hinted at in exercise 3.11 above. We use the concept of *value safety* to describe how a compiler is allowed to change the interpretation of a computation. At its strictest, the compiler is not allowed to make any changes that affect the result of a computation.

Compilers typically have an option controlling whether optimizations are allowed that may change the numerical behavior. For the Intel compiler that is `-fp-model=...` On the other hand, options such as `-Ofast` are aimed at performance improvement only, and may affect numerical behavior severely. For the Gnu compiler full 754 compliance takes the option `-frounding-math` whereas `-ffast-math` allows for performance-oriented compiler transformations that violate 754 and/or the language standard.

These matters are also of importance if you care about *reproducibility* of results. If a code is compiled with two different compilers, should runs with the same input give the same output? If a code is run in parallel on two different processor configurations? These questions are very subtle. In the first case, people sometimes insist on *bitwise reproducibility*, whereas in the second case some differences are allowed, as long as the result stays 'scientifically' equivalent. Of course, that concept is hard to make rigorous.

Here are some issues that are relevant when considering the influence of the compiler on code behavior and reproducibility.

4. This section borrows from documents by Microsoft http://msdn.microsoft.com/en-us/library/aa289157(vs.71).aspx and Intel http://software.intel.com/sites/default/files/article/164389/fp-consistency-122712_1.pdf; for detailed discussion the reader is referred to these.

Re-association Foremost among changes that a compiler can make to a computation is *re-association*, the technical term for grouping $a + b + c$ as $a + (b + c)$. The *C language standard* and the *C++ language standard* prescribe strict left-to-right evaluation of expressions without parentheses, so re-association is in fact not allowed by the standard. The *Fortran language standard* has no such prescription, but there the compiler has to respect the evaluation order that is implied by parentheses.

A common source of re-association is *loop unrolling*; see section 6.3.2. Under strict value safety, a compiler is limited in how it can unroll a loop, which has implications for performance. The amount of loop unrolling, and whether it's performed at all, depends on the compiler optimization level, the choice of compiler, and the target platform.

A more subtle source of re-association is parallel execution; see section 3.6.5. This implies that the output of a code need not be strictly reproducible between two runs on different parallel configurations.

Constant expressions It is a common compiler optimization to compute constant expressions during compile time. For instance, in

```
float one = 1.;
...
x = 2. + y + one;
```

the compiler change the assignment to `x = y+3.`. However, this violates the re-association rule above, and it ignores any dynamically set rounding behavior.

Expression evaluation In evaluating the expression $a+(b+c)$, a processor will generate an intermediate result for $b+c$ which is not assigned to any variable. Many processors are able to assign a higher *precision of the intermediate result*. A compiler can have a flag to dictate whether to use this facility.

Behavior of the floating point unit Rounding behavior (truncate versus round-to-nearest) and treatment of *gradual underflow* may be controlled by library functions or compiler options.

Library functions The IEEE 754 standard only prescribes simple operations; there is as yet no standard that treats sine or log functions. Therefore, their implementation may be a source of variability.

For more discussion, see [139].

3.8 More about floating point arithmetic

3.8.1 Kahan summation

The example in section 3.5.5 made visible some of the problems of computer arithmetic: rounding can cause results that are quite wrong, and very much dependent on evaluation order. A number of algorithms exist that try to compensate for these problems, in particular in the case of addition. We briefly discuss

Kahan summation[116], named after *William Kahan*, which is one example of a *compensated summation* algorithm.

```
sum ← 0
correction ← 0
while there is another input do
    oldsum ← sum
    input ← input − correction
    sum ← oldsum + input
    correction ← (sum − oldsum) − input
```

Exercise 3.26. Go through the example in section 3.5.5, adding a final term 3; that is compute $4 + 6 + 7 + 3$ and $6 + 7 + 4 + 3$ under the conditions of that example. Show that the correction is precisely the 3 undershoot when 17 is rounded to 20, or the 4 overshoot when 14 is rounded to 10; in both cases the correct result of 20 is computed.

3.8.2 Other computer arithmetic systems

Other systems have been proposed to dealing with the problems of inexact arithmetic on computers. One solution is *extended precision* arithmetic, where numbers are stored in more bits than usual. A common use of this is in the calculation of inner products of vectors: the accumulation is internally performed in extended precision, but returned as a regular floating point number. Alternatively, there are libraries such as GMPlib [76] that allow for any calculation to be performed in higher precision.

Another solution to the imprecisions of computer arithmetic is 'interval arithmetic' [113], where for each calculation interval bounds are maintained. While this has been researched for considerable time, it is not practically used other than through specialized libraries [21].

There have been some experiments with *ternary arithmetic* (see http://en.wikipedia.org/wiki/Ternary_computer and http://www.computer-museum.ru/english/setun.htm), however, no practical hardware exists.

3.8.3 Extended precision

When the IEEE 754 standard was drawn up, it was envisioned that processors could have a whole range of precisions. In practice, only single and double precision as defined have been used. However, one instance of *extended precision* still survives: Intel processors have 80-bit registers for storing intermediate results. (This goes back to the *Intel 8087 co-processor*.) This strategy makes sense in *FMA* instructions, and in the accumulation of inner products.

These 80-bit registers have a strange structure with an *significand integer* bit that can give rise to bit patterns that are not a valid representation of any defined number [165].

3.8.4 Reduced precision

You can ask 'does double precision always give benefits over single precision' and the answer is not always 'yes' but rather: 'it depends'.

3.8.4.1 *Lower precision in iterative refinement*

In iterative linear system solving (section 5.5, the accuracy is determined by how precise the residual is calculated, not how precise the solution step is done. Therefore, one could do operations such as applying the preconditioner (section 5.5.6) in reduced precision [26]. This is a form of *iterative refinement*; see section 5.5.6.

3.8.4.2 *Lower precision in Deep Learning*

IEEE 754-2008 has a definition for the *binary16* half precision format, which has a 5-bit exponent and 11-bit mantissa.

In *Machine Learning (ML)* it is more important to express the range of values than to be precise about the exact value. (This is the opposite of traditional scientific applications, where close values need to be resolved.) This has led to the definition of the *bfloat16* 'brain float' format [13], which is a 16-bit floating point format. It uses 8 bits for the exponent and 7 bits for the mantissa. This means that it shares the same exponent range as the IEEE single precision format; see figure 3.4.

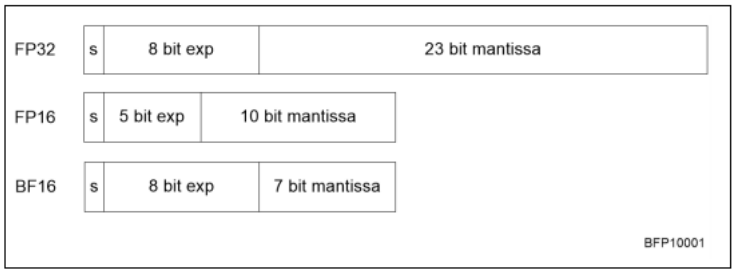


Figure 3.4: Comparison of fp32, fp16, and bfloat16 formats. (Illustration from [36].)

First of all, conversion between bfloat16 and fp32 is simple.

- Since bfloat16 and fp32 have the same structure in the first two bytes, a bfloat16 number can be derived from an fp32 number by truncating the third and fourth byte. However, rounding may give better results in practice.
- Conversely, casting a bloat16 to fp32 only requires filling the final two bytes with zeros.

A second argument for using bfloat16 is that it effectively doubles the available bandwidth. Thus, we see proposals for using this format where bandwdith is a limiting factor, and converting to regular formats elsewhere.

The limited precision of bfloat16 is probably enough to represent quantities in *DL* applications, but in order not to lose further precision it is envisioned that FMA hardware uses 32-bit numbers internally: the product of two bfloat16 number is a regular 32-bit number. In order to compute inner products (which happens as part of matrix-matrix multiplication in *DL*), we then need an FMA unit as in figure 3.5.

- The *Intel Knights Mill*, based on the *Intel Knights Landing*, has support for reduced precision.
- The *Intel Cooper Lake* implements the *bfloat16* format [36].

Even further reduction to 8-bit was discussed in [47].

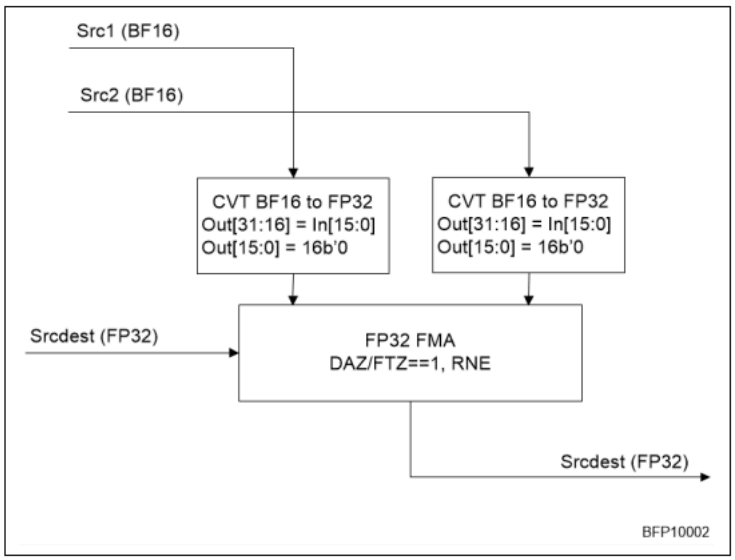


Figure 3.5: An FMA unit taking two bloat16 and one fp32 number. (Illustration from [36].)

3.8.5 Fixed-point arithmetic

A fixed-point number (for a more thorough discussion than found here, see [200]) can be represented as $\langle N, F\rangle$ where $N \geq \beta^0$ is the integer part and $F < 1$ is the fractional part. Another way of looking at this, is that a fixed-point number is an integer stored in $N + F$ digits, with an implied decimal point after the first N digits.

Fixed-point calculations can overflow, with no possibility to adjust an exponent. Consider the multiplication $\langle N_1, F_1\rangle \times \langle N_2, F_2\rangle$, where $N_1 \geq \beta^{n_1}$ and $N_2 \geq \beta^{n_2}$. This overflows if $n_1 + n_2$ is more than the number of positions available for the integer part. (Informally, the number of digits of the product is the sum of the number of digits of the operands.) This means that, in a program that uses fixed-point, numbers will need to have a number of leading zero digits, if you are ever going to multiply them, which lowers the numerical accuracy. It also means that the programmer has to think harder about calculations, arranging them in such a way that overflow will not occur, and that numerical accuracy is still preserved to a reasonable extent.

So why would people use fixed-point numbers? One important application is in embedded low-power devices, think a battery-powered digital thermometer. Since fixed-point calculations are essentially identical to integer calculations, they do not require a floating-point unit, thereby lowering chip size and lessening power demands. Also, many early video game systems had a processor that either had no floating-point unit, or where the integer unit was considerably faster than the floating-point unit. In both cases, implementing non-integer calculations as fixed-point, using the integer unit, was the key to high throughput.

Another area where fixed point arithmetic is still used is in signal processing. In modern CPUs, integer and floating point operations are of essentially the same speed, but converting between them is relatively slow. Now, if the sine function is implemented through table lookup, this means that in $\sin(\sin x)$ the output of a function is used to index the next function application. Obviously, outputting the sine function in fixed

point obviates the need for conversion between real and integer quantities, which simplifies the chip logic needed, and speeds up calculations.

3.8.6 Complex numbers

Some programming languages have *complex numbers* as a built-in data type, others not, and others are in between. For instance, in Fortran you can declare

```
COMPLEX z1,z2, z(32)
COMPLEX*16 zz1, zz2, zz(36)
```

A complex number is a pair of real numbers, the real and imaginary part, allocated adjacent in memory. The first declaration then uses 8 bytes to store to `REAL*4` numbers, the second one has `REAL*8`s for the real and imaginary part. (Alternatively, use `DOUBLE COMPLEX` or in Fortran90 `COMPLEX(KIND=2)` for the second line.)

By contrast, the C language does not directly have complex numbers, but both `C99` and `C++` have a `complex.h` header file[5] . This defines as complex number as in Fortran, as two real numbers.

Storing a complex number like this is easy, but sometimes it is computationally not the best solution. This becomes apparent when we look at arrays of complex numbers. If a computation often relies on access to the real (or imaginary) parts of complex numbers exclusively, striding through an array of complex numbers, has a stride two, which is disadvantageous (see section 1.3.5.8). In this case, it is better to allocate one array for the real parts, and another for the imaginary parts.

Exercise 3.27. Suppose arrays of complex numbers are stored the Fortran way. Analyze the memory access pattern of pairwise multiplying the arrays, that is, $\forall_i : c_i \leftarrow a_i \cdot b_i$, where `a(), b(), c()` are arrays of complex numbers.

Exercise 3.28. Show that an $n \times n$ linear system $Ax = b$ over the complex numbers can be written as a $2n \times 2n$ system over the real numbers. Hint: split the matrix and the vectors in their real and imaginary parts. Argue for the efficiency of storing arrays of complex numbers as separate arrays for the real and imaginary parts.

3.9 Conclusions

Computations done on a computer are invariably beset with numerical error. In a way, the reason for the error is the imperfection of computer arithmetic: if we could calculate with actual real numbers there would be no problem. (There would still be the matter of measurement error in data, and approximations made in numerical methods; see the next chapter.) However, if we accept roundoff as a fact of life, then various observations hold:

- Mathematically equivalent operations need not behave identically from a point of stability; see the 'abc-formula' example.
- Even rearrangements of the same computations do not behave identically; see the summing example.

5. These two header files are not identical, and in fact not compatible. Beware, if you compile C code with a C++ compiler [52].

Thus it becomes imperative to analyze computer algorithms with regard to their roundoff behavior: does roundoff increase as a slowly growing function of problem parameters, such as the number of terms evaluated, or is worse behavior possible? We will not address such questions in further detail in this book.

3.10 Review questions

Exercise 3.29. True or false?

- For integer types, the 'most negative' integer is the negative of the 'most positive' integer.
- For floating point types, the 'most negative' number is the negative of the 'most positive' one.
- For floating point types, the smallest positive number is the reciprocal of the largest positive number.

Chapter 4

Numerical treatment of differential equations

In this chapter we will look at the numerical solution of Ordinary Diffential Equations (ODEs) and Partial Diffential Equations (PDEs). These equations are commonly used in physics to describe phenomena such as the flow of air around an aircraft, or the bending of a bridge under various stresses. While these equations are often fairly simple, getting specific numbers out of them ('how much does this bridge sag if there are a hundred cars on it') is more complicated, often taking large computers to produce the desired results. Here we will describe the techniques that turn ODEs and PDEs into computable problems, that is, linear algebra. Chapter 5 will then look at computational aspects of these linear algebra problems.

First of all, in section 4.1 we will look at Initial Value Problems (IVPs), which describes processes that develop in time. Here we consider ODEs: scalar functions that are only depend on time. The name derives from the fact that typically the function is specified at an initial time point.

Next, in section 4.2 we will look at Boundary Value Problems (BVPs), describing processes in space. In realistic situations, this will concern multiple space variables, so we have a PDE. The name BVP is explained by the fact that the solution is specified on the boundary of the domain of definition.

Finally, in section 4.3 we will consider the 'heat equation', an Initial Boundary Value Problem (IBVP) which has aspects of both IVPs and BVPs: it describes heat spreading through a physical object such as a rod. The initial value describes the initial temperature, and the boundary values give prescribed temperatures at the ends of the rod.

Our aim in this chapter is to show the origins of an important class of computational problems. Therefore we will not go into theoretical matters of existence, uniqueness, or conditioning of solutions. For this, see [97] or any book that is specifically dedicated to ODEs or PDEs. For ease of analysis we will also assume that all functions involved have sufficiently many higher derivatives, and that each derivative is sufficiently smooth.

4.1 Initial value problems

Many physical phenomena change over time, and typically the laws of physics give a description of the change, rather than of the quantity of interest itself. For instance, Newton's second law

$$F = ma \tag{4.1}$$

is a statement about the change in position of a point mass: expressed as

$$a(t) = \frac{d^2}{dt^2}x(t) = F/m \tag{4.2}$$

it states that acceleration depends linearly on the force exerted on the mass. A closed form description $x(t) = \ldots$ for the location of the mass can sometimes be derived analytically, but in many cases some approximation, usually by numerical computation, is needed. This is also known as 'numerical integration'.

Newton's second law (4.1) is an ODE since it describes a function of one variable, time. It is an IVP since it describes the development in time, starting with some initial conditions. As an ODE, it is 'of second order' since it involves a second derivative, We can reduce this to first order, involving only first derivatives, if we allow vector quantities. We introduce a two-component vector u that combines the location x and velocity x', where we use the prime symbol to indicate differentiation in case of functions of a single variable:

$$u(t) = (x(t), x'(t))^t. \tag{4.3}$$

Newton's equation expressed in u then becomes:

$$u' = Au + B, \qquad A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 0 \\ F/a \end{pmatrix}. \tag{4.4}$$

For simplicity, in this course we will only consider first-order scalar equations; our reference equation is then

$$u'(t) = f(t, u(t)), \qquad u(0) = u_0, \qquad t > 0. \tag{4.5}$$

Equation (4.5) allows for an explicit time dependence of the process, but in general we only consider equations that do not have this explicit dependence, the so-called 'autonomous' ODEs of the form

$$u'(t) = f(u(t)) \tag{4.6}$$

in which the right hand side does not explicitly depend on t.

Remark *Non-autonomous ODEs can be transformed to autonomous ones, so this is no limitation. If $u = u(t)$ is a scalar function and $f = f(t, u)$, we define $u_2(t) = t$ and consider the equivalent autonomous system* $\binom{u'}{u_2'} = \binom{f(u_2,u)}{1}$.

Typically, the initial value in some starting point (often chosen as $t = 0$) is given: $u(0) = u_0$ for some value u_0, and we are interested in the behavior of u as $t \to \infty$. As an example, $f(x) = x$ reduces equation (4.5) to $u'(t) = u(t)$. This is a simple model for population growth: the equation states that the rate of growth is equal to the size of the population. Now we consider the numerical solution and the accuracy of this process.

In a numerical method, we choose discrete size time steps to approximate the solution of the continuous time-dependent process. Since this introduces a certain amount of error, we will analyze the error introduced in each time step, and how this adds up to a global error. In some cases, the need to limit the global error will impose restrictions on the numerical scheme.

4.1.1 Error and stability

Since numerical computation will always involve inaccuracies stemming from the use of machine arithmetic, we want to avoid the situation where a small perturbation in the initial value leads to large perturbations in the solution. Therefore, we will call a differential equation 'stable' if solutions corresponding to different (but close enough) initial values u_0 converge to one another as $t \to \infty$.

Theorem 2 *A sufficient criterion for stability is:*

$$\frac{\partial}{\partial u} f(u) = \begin{cases} > 0 & \textit{unstable} \\ = 0 & \textit{neutrally stable} \\ < 0 & \textit{stable} \end{cases} \tag{4.7}$$

Proof. If u^* is a zero of f, meaning $f(u^*) = 0$, then the constant function $u(t) \equiv u^*$ is a solution of $u' = f(u)$, a so-called 'equilibrium' solution. We will now consider how small perturbations from the equilibrium behave. Let u be a solution of the PDE, and write $u(t) = u^* + \eta(t)$, then we have

$$\begin{aligned} \eta' = u' &= f(u) = f(u^* + \eta) = f(u^*) + \eta f'(u^*) + O(\eta^2) \\ &= \eta f'(u^*) + O(\eta^2) \end{aligned} \tag{4.8}$$

Ignoring the second order terms, this has the solution

$$\eta(t) = e^{f'(x^*)t} \tag{4.9}$$

which means that the perturbation will damp out if $f'(x^*) < 0$, and amplify if $f'(x^*) > 0$.

We will often refer to the simple example $f(u) = -\lambda u$, with solution $u(t) = u_0 e^{-\lambda t}$. This problem is stable if $\lambda > 0$.

4.1.2 Finite difference approximation: explicit and implicit Euler methods

In order to solve ODEs numerically, we turn the continuous problem into a discrete one by looking at finite time/space steps. Assuming all functions are sufficiently smooth, a straightforward *Taylor series* expansion of u around t (see appendix 17) gives:

$$u(t + \Delta t) = u(t) + u'(t)\Delta t + u''(t)\frac{\Delta t^2}{2!} + u'''(t)\frac{\Delta t^3}{3!} + \cdots \tag{4.10}$$

This gives for u':

$$\begin{aligned} u'(t) &= \frac{u(t+\Delta t) - u(t)}{\Delta t} + \frac{1}{\Delta t}\left(u''(t)\frac{\Delta t^2}{2!} + u'''(t)\frac{\Delta t^3}{3!} + \cdots \right) \\ &= \frac{u(t+\Delta t) - u(t)}{\Delta t} + \frac{1}{\Delta t} O(\Delta t^2) \\ &= \frac{u(t+\Delta t) - u(t)}{\Delta t} + O(\Delta t) \end{aligned} \tag{4.11}$$

We can approximate the infinite sum of higher derivatives by a single $O(\Delta t^2)$ if all derivatives are bounded; alternatively, you can show that this sum is equal to $\Delta t^2 u''(t + \alpha \Delta t)$ with $0 < \alpha < 1$.

We see that we can approximate a differential operator by a *finite difference*, with an error that is known in its order of magnitude as a function of the time step.

Substituting this in $u' = f(t,u)$ transforms equation 4.6 into

$$\frac{u(t+\Delta t) - u(t)}{\Delta t} = f(t,u(t)) + O(\Delta t) \tag{4.12}$$

or

$$u(t+\Delta t) = u(t) + \Delta t\, f(t,u(t)) + O(\Delta t^2). \tag{4.13}$$

Remark *The preceding two equations are mathematical equalities, and should not be interpreted as a way of computing u' for a given function u. Recalling the discussion in section 3.5.4 you can see that such a formula would quickly lead to cancellation for small Δt. Further discussion of numerical differentiation is outside the scope of this book; please see any standard numerical analysis textbook.*

We now use equation (4.13) to derive a numerical scheme: with $t_0 = 0$, $t_{k+1} = t_k + \Delta t = \cdots = (k+1)\Delta t$, we get a difference equation

$$u_{k+1} = u_k + \Delta t\, f(t_k, u_k) \tag{4.14}$$

for u_k quantities, and we hope that u_k will be a good approximation to $u(t_k)$. This is known as the *Explicit Euler method*, or *Euler forward method*.

The process of going from a differential equation to a difference equation is often referred to as *discretization*, since we compute function values only in a discrete set of points. The values computed themselves are still real valued. Another way of phrasing this: the numerical solution is found in the finite dimensional space $\mathbb{R}^k$ if we compute k time steps. The solution to the original problem is in the infnite-dimensional space of functions $\mathbb{R} \to \mathbb{R}$.

In equation (4.11) we approximated one operator by another, and in doing so made a *truncation error* of order $O(\Delta t)$ as $\Delta t \downarrow 0$ (see appendix 15 for a more formal introduction to this notation for orders of magnitude.). This does *not* immediately imply that the difference equation computes a solution that is close to the true solution. For that some more analysis is needed.

We start by analyzing the 'local error': if we assume the computed solution is exact at step k, that is, $u_k = u(t_k)$, how wrong will we be at step $k+1$? We have

$$\begin{aligned} u(t_{k+1}) &= u(t_k) + u'(t_k)\Delta t + u''(t_k)\frac{\Delta t^2}{2!} + \cdots \\ &= u(t_k) + f(t_k, u(t_k))\Delta t + u''(t_k)\frac{\Delta t^2}{2!} + \cdots \end{aligned} \tag{4.15}$$

and

$$u_{k+1} = u_k + f(t_k u_k)\Delta t \tag{4.16}$$

So

$$\begin{aligned} L_{k+1} &= u_{k+1} - u(t_{k+1}) = u_k - u(t_k) + f(t_k, u_k) - f(t_k, u(t_k)) - u''(t_k)\tfrac{\Delta t^2}{2!} + \cdots \\ &= -u''(t_k)\tfrac{\Delta t^2}{2!} + \cdots \end{aligned} \tag{4.17}$$

This shows that in each step we make an error of $O(\Delta t^2)$. If we assume that these errors can be added, we find a global error of

$$E_k \approx \Sigma_k L_k = k\Delta t \frac{\Delta t^2}{2!} = O(\Delta t) \tag{4.18}$$

Since the global error is of first order in Δt, we call this a 'first order method'. Note that this error, which measures the distance between the true and computed solutions, is of the same order $O(\Delta t)$ as the truncation error, which is the error in approximating the operator.

4.1.2.1 Stability of the Explicit Euler method

Consider the IVP $u' = f(t,u)$ for $t \geq 0$, where $f(t,u) = -\lambda u$ and an initial value $u(0) = u_0$ is given. This has an exact solution of $u(t) = u_0 e^{-\lambda t}$. From the above discussion, we conclude that this problem is stable, meaning that small perturbations in the solution ultimately damp out, if $\lambda > 0$. We will now investigate the question of whether the numerical solution behaves the same way as the exact solution, that is, whether numerical solutions also converge to zero.

The Euler forward, or explicit Euler, scheme for this problem is

$$u_{k+1} = u_k - \Delta t \lambda u_k = (1 - \lambda\Delta t)u_k \Rightarrow u_k = (1 - \lambda\Delta t)^k u_0. \tag{4.19}$$

For stability, we require that $u_k \to 0$ as $k \to \infty$. This is equivalent to

$$\begin{aligned} u_k \downarrow 0 &\Leftrightarrow |1 - \lambda\Delta t| < 1 \\ &\Leftrightarrow -1 < 1 - \lambda\Delta t < 1 \\ &\Leftrightarrow -2 < -\lambda\Delta t < 0 \\ &\Leftrightarrow 0 < \lambda\Delta t < 2 \\ &\Leftrightarrow \Delta t < 2/\lambda \end{aligned}$$

We see that the *stability* of the numerical solution scheme depends on the value of Δt: the scheme is only stable if Δt is small enough. For this reason, we call the explicit Euler method *conditionally stable*. Note that the stability of the differential equation and the stability of the numerical scheme are two different questions. The continuous problem is stable if $\lambda > 0$; the numerical problem has an additional condition that depends on the discretization scheme used.

Note that the stability analysis we just performed was specific to the differential equation $u' = -\lambda u$. If you are dealing with a different IVP you have to perform a separate analysis. However, you will find that explicit methods typically give conditional stability.

4.1.2.2 The Euler implicit method

The explicit method you just saw was easy to compute, but the conditional stability is a potential problem. For instance, it could imply that the number of time steps would be a limiting factor. There is an alternative to the explicit method that does not suffer from the same objection.

Instead of expanding $u(t + \Delta t)$, consider the following expansion of $u(t - \Delta t)$:

$$u(t - \Delta t) = u(t) - u'(t)\Delta t + u''(t)\frac{\Delta t^2}{2!} + \cdots \tag{4.20}$$

which implies

$$u'(t) = \frac{u(t) - u(t - \Delta t)}{\Delta t} + u''(t)\Delta t/2 + \cdots \tag{4.21}$$

As before, we take the equation $u'(t) = f(t, u(t))$ and approximate $u'(t)$ by a difference formula:

$$\frac{u(t) - u(t - \Delta t)}{\Delta t} = f(t, u(t)) + O(\Delta t) \Rightarrow u(t) = u(t - \Delta t) + \Delta t f(t, u(t)) + O(\Delta t^2) \tag{4.22}$$

Again we define fixed points $t_k = kt$, and we define a numerical scheme:

$$u_{k+1} = u_k + \Delta t f(t_{k+1}, u_{k+1}) \tag{4.23}$$

where u_k is an approximation of $u(t_k)$.

An important difference with the explicit scheme is that u_{k+1} now also appears on the right hand side of the equation. That is, the computation of u_{k+1} is now implicit. For example, let $f(t, u) = -u^3$, then $u_{k+1} = u_k - \Delta t u_{k+1}^3$. In other words, u_{k+1} is the solution for x of the equation $\Delta t x^3 + x = u_k$. This is a nonlinear equation, which typically can be solved using the Newton method.

4.1.2.3 Stability of the implicit Euler method

Let us take another look at the example $f(t, u) = -\lambda u$. Formulating the implicit method gives

$$u_{k+1} = u_k - \lambda \Delta t u_{k+1} \Leftrightarrow (1 + \lambda \Delta t) u_{k+1} = u_k \tag{4.24}$$

so

$$u_{k+1} = \left(\frac{1}{1 + \lambda \Delta t}\right) u_k \Rightarrow u_k = \left(\frac{1}{1 + \lambda \Delta t}\right)^k u_0. \tag{4.25}$$

If $\lambda > 0$, which is the condition for a stable equation, we find that $u_k \to 0$ for all values of λ and Δt. This method is called *unconditionally stable*. One advantage of an implicit method over an explicit one is clearly the stability: it is possible to take larger time steps without worrying about unphysical behavior. Of course, large time steps can make convergence to the *steady state* (see Appendix 16.4) slower, but at least there will be no divergence.

On the other hand, implicit methods are more complicated. As you saw above, they can involve nonlinear systems to be solved in every time step. In cases where u is vector-valued, such as in the heat equation, discussed below, you will see that the implicit method requires the solution of a system of equations.

Exercise 4.1. Analyze the accuracy and computational aspects of the following scheme for the IVP $u'(x) = f(x)$:

$$u_{i+1} = u_i + h(f(x_i) + f(x_{i+1}))/2 \tag{4.26}$$

which corresponds to adding the Euler explicit and implicit schemes together. You do not have to analyze the stability of this scheme.

Exercise 4.2. Consider the initial value problem $y'(t) = y(t)(1-y(t))$. Observe that $y \equiv 0$ and $y \equiv 1$ are solutions. These are called 'equilibrium solutions'.

1. A solution is stable, if perturbations 'converge back to the solution', meaning that for ϵ small enough,

   $$\text{if } y(t) = \epsilon \text{ for some } t, \text{ then } \lim_{t\to\infty} y(t) = 0 \tag{4.27}$$

   and

   $$\text{if } y(t) = 1 + \epsilon \text{ for some } t, \text{ then } \lim_{t\to\infty} y(t) = 1 \tag{4.28}$$

   This requires for instance that

   $$y(t) = \epsilon \Rightarrow y'(t) < 0. \tag{4.29}$$

   Investigate this behavior. Is zero a stable solution? Is one?
2. Consider the explicit method

   $$y_{k+1} = y_k + \Delta t y_k(1 - y_k) \tag{4.30}$$

   for computing a numerical solution to the differential equation. Show that

   $$y_k \in (0,1) \Rightarrow y_{k+1} > y_k, \qquad y_k > 1 \Rightarrow y_{k+1} < y_k \tag{4.31}$$

3. Write a small program to investigate the behavior of the numerical solution under various choices for Δt. Include program listing and a couple of runs in your homework submission.
4. You see from running your program that the numerical solution can oscillate. Derive a condition on Δt that makes the numerical solution monotone. It is enough to show that $y_k < 1 \Rightarrow y_{k+1} < 1$, and $y_k > 1 \Rightarrow y_{k+1} > 1$.
5. Now consider the implicit method

   $$y_{k+1} - \Delta t y_{k+1}(1 - y_{k+1}) = y_k \tag{4.32}$$

   and show that y_{k+1} can be computed from y_k. Write a program, and investigate the behavior of the numerical solution under various choices for Δt.
6. Show that the numerical solution of the implicit scheme is monotone for all choices of Δt.

4.2 Boundary value problems

In the previous section you saw initial value problems, which model phenomena that evolve over time. We will now move on to'boundary value problems', which are in general stationary in time, but which describe a phenomenon that is location dependent. Examples would be the shape of a bridge under a load, or the heat distribution in a window pane, as the temperature outside differs from the one inside.

The general form of a (second order, one-dimensional) BVP is

$$u''(x) = f(x, u, u') \text{ for } x \in [a, b] \text{ where } u(a) = u_a, u(b) = u_b \tag{4.33}$$

but here we will only consider the simple form (see appendix 16)

$$-u''(x) = f(x) \text{ for } x \in [0, 1] \text{ with } u(0) = u_0, u(1) = u_1. \tag{4.34}$$

in one space dimension, or

$$-u_{xx}(\bar{x}) - u_{yy}(\bar{x}) = f(\bar{x}) \text{ for } \bar{x} \in \Omega = [0, 1]^2 \text{ with } u(\bar{x}) = u_0 \text{ on } \delta\Omega. \tag{4.35}$$

in two space dimensions. Here, $\delta\Omega$ is the boundary of the domain Ω. Since we prescribe the value of u on the boundary, such a problem is called a Boundary Value Problem (BVP).

Remark *The boundary conditions can be more general, involving derivatives on the interval end points. Here we only look at* Dirichlet boundary conditions *which prescribe function values on the boundary of the domain. The difference with* Neumann boundary conditions *are more mathematical than computational.*

4.2.1 General PDE theory

There are several types of PDE, each with distinct mathematical properties. The most important property is that of *region of influence*: if we tinker with the problem so that the solution changes in one point, what other points will be affected.

4.2.1.1 Hyperbolic equations

PDEs are of the form

$$Au_{xx} + Bu_{yy} + \text{lower order terms} = 0 \tag{4.36}$$

with A, B of opposite sign. Such equations describe waves, or more general convective phenomena, that are conservative, and do not tend to a steady state.

Intuitively, changing the solution of a wave equation at any point will only change certain future points, since waves have a propagation speed that makes it impossible for a point to influence points in the near future that are too far away in space. This type of PDE will not be discussed in this book.

4.2.1.2 *Parabolic equations*

PDEs are of the form

$$Au_x + Bu_{yy} + \text{no higher order terms in } x = 0 \tag{4.37}$$

and they describe diffusion-like phenomena; these often tend to a *steady state*. The best way to characterize them is to consider that the solution in each point in space and time is influenced by a certain finite region at each previous point in space.

Remark *This leads to a condition limiting the time step in* IBVP*, known as the* Courant-Friedrichs-Lewy condition*. It describes the notion that in the exact problem $u(x,t)$ depends on a range of $u(x',t-\Delta t)$ values; the time step of the numerical method has to be small enough that the numerical solution takes all these points into account.*

The *heat equation* (section 4.3) is the standard example of the parabolic type.

4.2.1.3 *Elliptic equations*

PDEs have the form

$$Au_{xx} + Bu_{yy} + \text{lower order terms} = 0 \tag{4.38}$$

where $A, B > 0$; they typically describe processes that have reached a *steady state*, for instance as $t \to \infty$ in a parabolic problem. They are characterized by the fact that all points influence each other. These equations often describe phenomena in structural mechanics, such as a beam or a membrane. It is intuitively clear that pressing down on any point of a membrane will change the elevation of every other point, no matter how little. The *Laplace equation* (section 4.2.2) is the standard example of this type.

4.2.2 The Laplace equation in one space dimension

We call the operator Δ, defined by

$$\Delta u = u_{xx} + u_{yy}, \tag{4.39}$$

a second order *differential operator*, and equation (4.35) a second-order PDE. Specifically, the problem

$$-\Delta u = -u_{xx}(\bar{x}) - u_{yy}(\bar{x}) = f(\bar{x}) \text{ for } \bar{x} \in \Omega = [0,1]^2 \text{ with } u(\bar{x}) = u_0 \text{ on } \delta\Omega. \tag{4.40}$$

is called the *Poisson equation*, in this case defined on the unit square. The case of $f \equiv 0$ is called the *Laplace equation*. Second order PDEs are quite common, describing many phenomena in fluid and heat flow and structural mechanics.

At first, for simplicity, we consider the one-dimensional Poisson equation

$$-u_{xx} = f(x). \tag{4.41}$$

We will consider the two-dimensional case below; the extension to three dimensions will then be clear.

In order to find a numerical scheme we use Taylor series as before, expressing $u(x+h)$ and $u(x-h)$ in terms of u and its derivatives at x. Let $h > 0$, then

$$u(x+h) = u(x) + u'(x)h + u''(x)\frac{h^2}{2!} + u'''(x)\frac{h^3}{3!} + u^{(4)}(x)\frac{h^4}{4!} + u^{(5)}(x)\frac{h^5}{5!} + \cdots \tag{4.42}$$

and

$$u(x-h) = u(x) - u'(x)h + u''(x)\frac{h^2}{2!} - u'''(x)\frac{h^3}{3!} + u^{(4)}(x)\frac{h^4}{4!} - u^{(5)}(x)\frac{h^5}{5!} + \cdots \tag{4.43}$$

Our aim is now to approximate $u''(x)$. We see that the u' terms in these equations would cancel out under addition, leaving $2u(x)$:

$$u(x+h) + u(x-h) = 2u(x) + u''(x)h^2 + u^{(4)}(x)\frac{h^4}{12} + \cdots \tag{4.44}$$

so

$$-u''(x) = \frac{2u(x) - u(x+h) - u(x-h)}{h^2} + u^{(4)}(x)\frac{h^2}{12} + \cdots \tag{4.45}$$

The basis for a numerical scheme for (4.34) is then the observation

$$\frac{2u(x) - u(x+h) - u(x-h)}{h^2} = f(x, u(x), u'(x)) + O(h^2), \tag{4.46}$$

which shows that we can approximate the differential operator by a difference operator, with an $O(h^2)$ *truncation error* as $h \downarrow 0$.

To derive a numerical method, we divide the interval $[0,1]$ into equally spaced points: $x_k = kh$ where $h = 1/(n+1)$ and $k = 0 \ldots n+1$. With these, the FD formula (4.45) leads to a numerical scheme that forms a system of equations:

$$-u_{k+1} + 2u_k - u_{k-1} = h^2 f(x_k) \quad \text{for } k = 1, \ldots, n \tag{4.47}$$

This process of using the FD formula (4.45) for the approximate solution of a PDE is known as the *Finite Difference Method (FDM).*

For most values of k this equation relates u_k unknown to the unknowns u_{k-1} and u_{k+1}. The exceptions are $k = 1$ and $k = n$. In that case we recall that u_0 and u_{n+1} are known boundary conditions, and we write the equations with unknowns on the left and known quantities on the right as

$$\begin{cases} -u_{i-1} + 2u_i - u_{i+1} = h^2 f(x_i) \\ 2u_1 - u_2 = h^2 f(x_1) + u_0 \\ 2u_n - u_{n-1} = h^2 f(x_n) + u_{n+1}. \end{cases} \tag{4.48}$$

We can now summarize these equations for $u_k, k = 1 \ldots n-1$ as a matrix equation:

$$\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & & -1 & 2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} h^2 f_1 + u_0 \\ h^2 f_2 \\ \vdots \\ h^2 f_n + u_{n+1} \end{pmatrix} \tag{4.49}$$

This has the form $Au = f$ with A a fully known matrix, f a fully known vector, and u a vector of unknowns. Note that the right hand side vector has the boundary values of the problem in the first and last locations. This means that, if you want to solve the same differential equation with different boundary conditions, only the vector f changes.

Exercise 4.3. A condition of the type $u(0) = u_0$ is called a *Dirichlet boundary condition*. Physically, this corresponds to, for instance, knowing the temperature at the end point of a rod. Other boundary conditions exist. Specifying a value for the derivative, $u'(0) = u'_0$, rather than for the function value,would be appropriate if we are modeling fluid flow and the outflow rate at $x = 0$ is known. This is known as a *Neumann boundary condition*.

A Neumann boundary condition $u'(0) = u'_0$ can be modeled by stating

$$\frac{u_0 - u_1}{h} = u'_0. \tag{4.50}$$

Show that, unlike in the case of the Dirichlet boundary condition, this affects the matrix of the linear system.

Show that having a Neumann boundary condition at both ends gives a singular matrix, and therefore no unique solution to the linear system. (Hint: guess the vector that has eigenvalue zero.)

Physically this makes sense. For instance, in an elasticity problem, Dirichlet boundary conditions state that the rod is clamped at a certain height; a Neumann boundary condition only states its angle at the end points, which leaves its height undetermined.

Let us list some properties of A that you will see later are relevant for solving such systems of equations:

- The matrix is very *sparse*: the percentage of elements that is nonzero is low. The nonzero elements are not randomly distributed but located in a band around the main diagonal. We call this a *banded matrix* in general, and a *tridiagonal matrix* in this specific case.
  The banded structure is typical for PDEs, but sparse matrices in different applications can be less regular.
- The matrix is symmetric. This property does not hold for all matrices that come from discretizing BVPs, but it is true if there are no odd order (meaning first, third, fifth,...) derivatives, such as u_x, u_{xxx}, u_{xy}.
- Matrix elements are constant in each diagonal, that is, in each set of points $\{(i, j) : i - j = c\}$ for some c. This is only true for very simple problems. It is no longer true if the differential equation has location dependent terms such as $\frac{d}{dx}(a(x)\frac{d}{dx}u(x))$. It is also no longer true if we make h variable through the interval, for instance because we want to model behavior around the left end point in more detail.
- Matrix elements conform to the following sign pattern: the diagonal elements are positive, and the off-diagonal elements are nonpositive. This property depends on the numerical scheme used; it is for instance true for second order schemes for second order equations without first order terms. Together with the following property of definiteness, this is called an *M-matrix*. There is a whole mathematical theory of these matrices [12].
- The matrix is positive definite: $x^t Ax > 0$ for all nonzero vectors x. This property is inherited from the original continuous problem, if the numerical scheme is carefully chosen. While the

use of this may not seem clear at the moment, later you will see methods for solving the linear system that depend on it.

Strictly speaking the solution of equation (4.49) is simple: $u = A^{-1}f$. However, computing A^{-1} is not the best way of finding u. As observed just now, the matrix A has only $3N$ nonzero elements to store. Its inverse, on the other hand, does not have a single nonzero. Although we will not prove it, this sort of statement holds for most sparse matrices. Therefore, we want to solve $Au = f$ in a way that does not require $O(n^2)$ storage.

Exercise 4.4. How would you solve the tridiagonal system of equations? Show that the *LU factorization* of the coefficient matrix gives factors that are of *bidiagonal matrix* form: they have a nonzero diagonal and exactly one nonzero sub or super diagonal.
What is the total number of operations of solving the tridiagonal system of equations? What is the operation count of multiplying a vector with such a matrix? This relation is not typical!

4.2.3 The Laplace equation in two space dimensions

The one-dimensional BVP above was atypical in a number of ways, especially related to the resulting linear algebra problem. In this section we will investigate the two-dimensional Poisson/Laplace problem. You'll see that it constitutes a non-trivial generalization of the one-dimensional problem. The three-dimensional case is very much like the two-dimensional case, so we will not discuss it. (For one essential difference, see the discussion in section 7.8.1.)

The one-dimensional problem above had a function $u = u(x)$, which now becomes $u = u(x, y)$ in two dimensions. The two-dimensional problem we are interested is then

$$-u_{xx} - u_{yy} = f, \quad (x, y) \in [0, 1]^2, \tag{4.51}$$

where the values on the boundaries are given. We get our discrete equation by applying equation (4.45) in x and y directions:

$$\begin{aligned} -u_{xx}(x, y) &= \frac{2u(x,y) - u(x+h,y) - u(x-h,y)}{h^2} + u^{(4)}(x, y)\frac{h^2}{12} + \cdots \\ -u_{yy}(x, y) &= \frac{2u(x,y) - u(x,y+h) - u(x,y-h)}{h^2} + u^{(4)}(x, y)\frac{h^2}{12} + \cdots \end{aligned} \tag{4.52}$$

or, taken together:

$$4u(x, y) - u(x + h, y) - u(x - h, y) - u(x, y + h) - u(x, y - h) = 1/h^2\, f(x, y) + O(h^2) \tag{4.53}$$

Let again $h = 1/(n + 1)$ and define $x_i = ih$ and $y_j = jh$; let u_{ij} be the approximation to $u(x_i, y_j)$, then our discrete equation becomes

$$4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} = h^{-2} f_{ij}. \tag{4.54}$$

We now have $n \times n$ unknowns u_{ij}. To render this in a *linear system* as before we need to put them in a linear ordering, which we do by defining $I = I_{ij} = j + i \times n$. This is called the *lexicographic ordering* since it sorts the coordinates (i, j) as if they are strings.

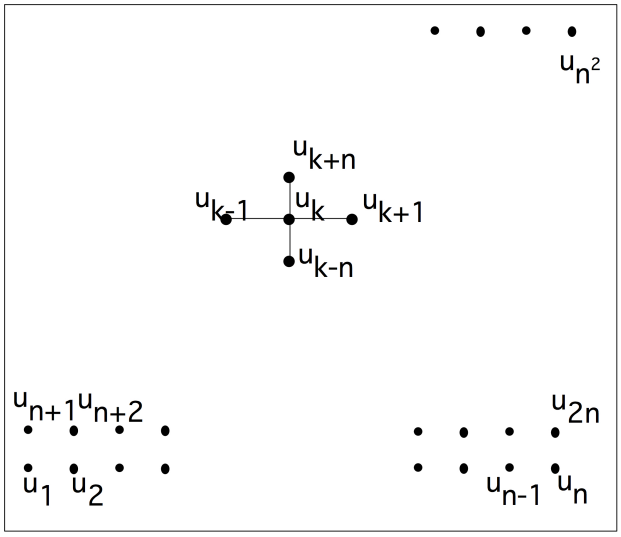


Figure 4.1: A difference stencil applied to a two-dimensional square domain.

Using this ordering gives us $N = n^2$ equations

$$4u_I - u_{I+1} - u_{I-1} - u_{I+n} - u_{I-n} = h^{-2} f_I, \qquad I = 1, \ldots, N \tag{4.55}$$

and the linear system looks like

$$A = \left(\begin{array}{cccccc|cccccc|cc}
4 & -1 & & & & \emptyset & -1 & & & & & \emptyset & & \\
-1 & 4 & -1 & & & & & -1 & & & & & & \\
 & \ddots & \ddots & \ddots & & & & & \ddots & & & & & \\
 & & \ddots & \ddots & -1 & & & & & \ddots & & & & \\
\emptyset & & & -1 & 4 & & \emptyset & & & & & -1 & & \\
\hline
-1 & & & & \emptyset & & 4 & -1 & & & & & -1 & \\
 & -1 & & & & & -1 & 4 & -1 & & & & & -1 \\
 & \uparrow & \ddots & & & & \uparrow & \uparrow & \uparrow & & & & & \uparrow \\
 & k-n & & & & & k-1 & k & k+1 & & -1 & & & k+n \\
 & & & & -1 & & & & & -1 & 4 & & & \\
\hline
 & & & & & & \ddots & & & & & & \ddots &
\end{array}\right) \tag{4.56}$$

where the matrix is of size $N \times N$, with $N = n^2$. As in the one-dimensional case, we see that a BVP gives rise to a *sparse matrix*.

It may seem the natural thing to consider this system of linear equations in its matrix form. However, it can be more insightful to render these equations in a way that makes clear the two-dimensional connections of the unknowns. For this, figure 4.1 pictures the variables in the domain, and how equation (4.54) relates them through a *finite difference stencil*. (See section 4.2.4 for more on stencils.) From now on, when making such a picture of the domain, we will just use the indices of the variables, and omit the 'u' identifier.

The matrix of equation 4.56 is banded as in the one-dimensional case, but unlike in the one-dimensional case, there are zeros inside the band. (This has some important consequences when we try to solve the

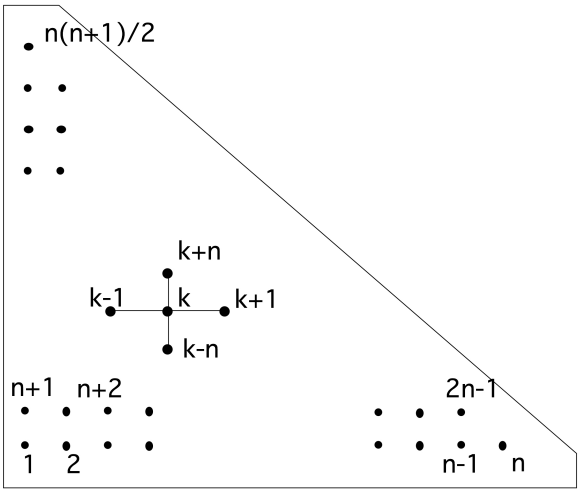


Figure 4.2: A triangular domain of definition for the Poisson equation.

linear system; see section 5.4.3.) Because the matrix has five nonzero diagonals, it is said to be of *penta-diagonal* structure.

You can also put a block structure on the matrix, by grouping the unknowns together that are in one row of the domain. This is called a *block matrix*, and, on the block level, it has a *tridiagonal matrix* structure, so we call this a *block tridiagonal* matrix. Note that the diagonal blocks themselves are tridiagonal; the off-diagonal blocks are minus the identity matrix.

This matrix, like the one-dimensional example above, has constant diagonals, but this is again due to the simple nature of the problem. In practical problems it will not be true. That said, such 'constant coefficient' problems occur, and when they are on rectangular domains, there are very efficient methods for solving the linear system with $N \log N$ time complexity.

Exercise 4.5. The block structure of the matrix, with all diagonal blocks having the same size, is due to the fact that we defined our BVP on a square domain. Sketch the matrix structure that arises from discretizing equation (4.51), again with central differences, but this time defined on a triangular domain; see figure 4.2. Show that, again, there is a block tridiagonal matrix structure, but that the blocks are now of varying sizes. Hint: start by sketching a small example. For $n = 4$ you should get a 10×10 matrix with a 4×4 block structure.

For domains that are even more irregular, the matrix structure will also be irregular.

The regular block structure is also caused by our decision to order the unknowns by rows and columns. This known as the *natural ordering* or *lexicographic ordering*; various other orderings are possible. One common way of ordering the unknowns is the *red-black ordering* or *checkerboard ordering* which has advantages for parallel computation. This will be discussed in section 7.7.

Remark *For this simple matrix it is possible to determine eigenvalue and eigenvectors by a* Fourier analysis. *In more complicated cases, such as an operator* $\nabla a(x,y)\nabla x$*, this is not possible, but the* Gershgorin theorem

will still tell us that the matrix is positive definite, or at least semi-definite. This corresponds to the continuous problem being coercive.

There is more to say about analytical aspects of the BVP (for instance, how smooth is the solution and how does that depend on the boundary conditions?) but those questions are outside the scope of this course. Here we only focus on the numerical aspects of the matrices. In the chapter on linear algebra, specifically sections 5.4 and 5.5, we will discuss solving the linear system from BVPs.

4.2.4 Difference stencils

The discretization (4.53) is often phrased as applying the *difference stencil*

$$\begin{array}{ccc} \cdot & -1 & \cdot \\ -1 & 4 & -1 \\ \cdot & -1 & \cdot \end{array} \tag{4.57}$$

to the function u. Given a physical domain, we apply the stencil to each point in that domain to derive the equation for that point. Figure 4.1 illustrates that for a square domain of $n \times n$ points. Connecting this figure with equation (4.56), you see that the connections in the same line give rise to the main diagonal and first upper and lower offdiagonal; the connections to the next and previous lines become the nonzeros in the off-diagonal blocks.

This particular stencil is often referred to as the '5-point star' or *five-point stencil*. There are other difference stencils; the structure of some of them are depicted in figure 4.3. A stencil with only connections in

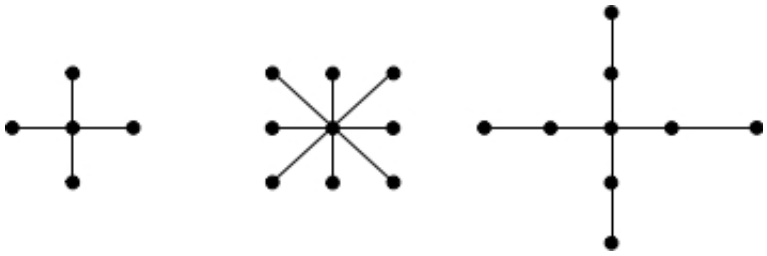


Figure 4.3: The structure of some difference stencils in two dimensions.

horizontal or vertical direction is called a 'star stencil', while one that has cross connections (such as the second in figure 4.3) is called a 'box stencil'.

Exercise 4.6. Consider the second and third stencil in figure 4.3, used for a BVP on a square domain. What does the sparsity structure of the resulting matrix look like, if we again order the variables by rows and columns?

Other stencils than the 5-point star can be used to attain higher accuracy, for instance giving a truncation error of $O(h^4)$. They can also be used for other differential equations than the one discussed above. For instance, it is not hard to show that for the equation $u_{xxxx} + u_{yyyy} = f$ we need a stencil that contains both $x, y \pm h$ and $x, y \pm 2h$ connections, such as the third stencil in the figure. Conversely, using the 5-point stencil no values of the coefficients give a discretization of the fourth order problem with less than $O(1)$ truncation error.

While the discussion so far has been about two-dimensional problems, it can be generalized to higher dimensions for such equations as $-u_{xx} - u_{yy} - u_{zz} = f$. The straightforward generalization of the 5-point stencil, for instance, becomes a *7-point stencil* in three dimensions.

Exercise 4.7. What does the matrix of of a central difference stencil in three dimensions look like? Describe the block structure.

4.2.5 Other discretization techniques

In the above, we used the FDM to find a numerical solution to a differential equation. There are various other techniques, and in fact, in the case of boundary value problems, they are usually preferred over finite differences. The most popular methods are the *FEM* and the *finite volume method*. Especially the finite element method is attractive, since it can handle irregular shapes more readily than finite differences, and it is more amenable to approximation error analysis. However, on the simple problems discussed here it gives similar or even the same linear systems as FD methods, so we limit the discussion to Finite Differences, since we are mostly concerned with the computational aspects of the linear systems.

There will be a brief discussion of finite element matrices in section 7.6.2.

4.3 Initial boundary value problem

We will now go on to discuss an Initial Boundary Value Problem (IBVP), which, as you may deduce from the name, combines aspects of IVPs and BVPs: they combine both a space and a time derivative. Here we will limit ourselves to one space dimension.

The problem we are considering is that of heat conduction in a rod, where $T(x,t)$ describes the temperature in location x at time t, for $x \in [a,b]$, $t > 0$. The so-called *heat equation* (see section 16.3 for a short derivation) is:

$$\frac{\partial}{\partial t}T(x,t) - \alpha\frac{\partial^2}{\partial x^2}T(x,t) = q(x,t) \tag{4.58}$$

where

- The initial condition $T(x,0) = T_0(x)$ describes the initial temperature distribution.
- The boundary conditions $T(a,t) = T_a(t)$, $T(b,t) = T_b(t)$ describe the ends of the rod, which can for instance be fixed to an object of a known temperature.
- The material the rod is made of is modeled by a single parameter $\alpha > 0$, the thermal diffusivity, which describes how fast heat diffuses through the material.
- The forcing function $q(x,t)$ describes externally applied heating, as a function of both time and place.
- The space derivative is the same operator we have studied before.

There is a simple connection between the IBVP and the BVP: if the boundary functions T_a and T_b are constant, and q does not depend on time, only on location, then intuitively T will converge to a *steady state*. The equation for this is $-\alpha u''(x) = q$, which we studied above.

We will now proceed to discuss the heat equation in one space dimension. After the above discussion of the Poisson equation in two and three dimensions, the extension of the heat equation to higher dimensions should pose no particular problem.

4.3.1 Discretization

We now discretize both space and time, by $x_{j+1} = x_j + \Delta x$ and $t_{k+1} = t_k + \Delta t$, with boundary conditions $x_0 = a$, $x_n = b$, and $t_0 = 0$. We write T_j^k for the numerical solution at $x = x_j, t = t_k$; with a little luck, this will approximate the exact solution $T(x_j, t_k)$.

For the space discretization we use the central difference formula (4.47):

$$\frac{\partial^2}{\partial x^2} T(x, t_k)\Big|_{x=x_j} \Rightarrow \frac{T_{j-1}^k - 2T_j^k + T_{j+1}^k}{\Delta x^2}. \tag{4.59}$$

For the time discretization we can use any of the schemes in section 4.1.2. We will investigate again the explicit and implicit schemes, with similar conclusions about the resulting stability.

4.3.1.1 *Explicit scheme*

With explicit time stepping we approximate the time derivative as

$$\frac{\partial}{\partial t} T(x_j, t)\Big|_{t=t_k} \Rightarrow \frac{T_j^{k+1} - T_j^k}{\Delta t}. \tag{4.60}$$

Taking this together with the central differences in space, we now have

$$\frac{T_j^{k+1} - T_j^k}{\Delta t} - \alpha \frac{T_{j-1}^k - 2T_j^k + T_{j+1}^k}{\Delta x^2} = q_j^k \tag{4.61}$$

which we rewrite as

$$T_j^{k+1} = T_j^k + \frac{\alpha \Delta t}{\Delta x^2}(T_{j-1}^k - 2T_j^k + T_{j+1}^k) + \Delta t q_j^k \tag{4.62}$$

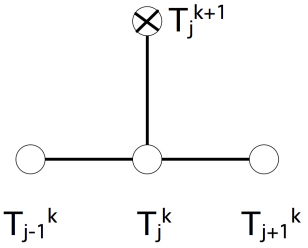


Figure 4.4: The difference stencil of the Euler forward method for the heat equation.

.

Pictorially, we render this as a difference stencil in figure 4.4. This expresses that the function value in each point is determined by a combination of points on the previous time level.

It is convenient to summarize the set of equations (4.62) for a given k and all values of j in vector form as

$$T^{k+1} = \left(I - \frac{\alpha \Delta t}{\Delta x^2} K\right) T^k + \Delta t q^k \tag{4.63}$$

where

$$K = \begin{pmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots \end{pmatrix}, \qquad T^k = \begin{pmatrix} T_1^k \\ \vdots \\ T_n^k \end{pmatrix}. \tag{4.64}$$

The important observation here is that the dominant computation for deriving the vector T^{k+1} from T^k is a simple matrix-vector multiplication:

$$T^{k+1} \leftarrow AT^k + \Delta t q^k \tag{4.65}$$

where $A = I - \frac{\alpha\Delta t}{\Delta x^2}K$. This is a first indication that the sparse matrix-vector product is an important operation; see sections 5.4 and 7.5. Actual computer programs using an explicit method often do not form the matrix, but evaluate the equation (4.62). However, the linear algebra formulation (4.63) is more insightful for purposes of analysis.

In later chapters we will consider parallel execution of operations. For now, we note that the explicit scheme is trivially parallel: each point can be updated with just the information of a few surrounding points.

4.3.1.2 *Implicit scheme*

In equation (4.60) we let T^{k+1} be defined from T^k. We can turn this around by defining T^k from T^{k-1}, as we did for the IVP in section 4.1.2.2. For the time discretization this gives

$$\frac{\partial}{\partial t}T(x,t)\Big|_{t=t_k} \Rightarrow \frac{T_j^k - T_j^{k-1}}{\Delta t} \qquad \text{or} \qquad \frac{\partial}{\partial t}T(x,t)\Big|_{t=t_{k+1}} \Rightarrow \frac{T_j^{k+1} - T_j^k}{\Delta t}. \tag{4.66}$$

The implicit time step discretization of the whole heat equation, evaluated in t_{k+1}, now becomes:

$$\frac{T_j^{k+1} - T_j^k}{\Delta t} - \alpha\frac{T_{j-1}^{k+1} - 2T_j^{k+1} + T_{j+1}^{k+1}}{\Delta x^2} = q_j^{k+1} \tag{4.67}$$

or

$$T_j^{k+1} - \frac{\alpha\Delta t}{\Delta x^2}(T_{j-1}^{k+1} - 2T_j^{k+1} + T_{j+1}^{k+1}) = T_j^k + \Delta t q_j^{k+1} \tag{4.68}$$

Figure 4.5 renders this as a stencil; this expresses that each point on the current time level influences a combination of points on the next level. Again we write this in vector form:

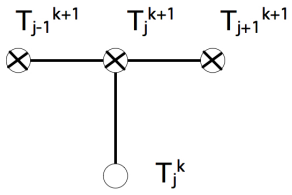


Figure 4.5: The difference stencil of the Euler backward method for the heat equation.

.

$$\left(I + \frac{\alpha\Delta t}{\Delta x^2}K\right)T^{k+1} = T^k + \Delta t q^{k+1} \tag{4.69}$$

As opposed to the explicit method, where a matrix-vector multiplication sufficed, the derivation of the vector T^{k+1} from T^k now involves a *linear system solution*:

$$T^{k+1} \leftarrow A^{-1}(T^k + \Delta t q^{k+1}) \tag{4.70}$$

where $A = I + \frac{\alpha\Delta t}{\Delta x^2}K$. Solving this linear system is a harder operation than the matrix-vector multiplication. Unlike what equation (4.70) suggests, codes using an implicit method do not actually form the inverse matrix, but rather solve the system (4.69) as such. Solving linear systems will be the focus of much of chapters 5 and 7.

In contrast to the explicit scheme, we now have no obvious parallelization strategy. The parallel solution of linear systems will occupy us in sections 7.6 and on.

Exercise 4.8. Show that the flop count for a time step of the implicit method is of the same order as of a time step of the explicit method. (This only holds for a problem with one space dimension.) Give at least one argument why we consider the implicit method as computationally 'harder'.

The numerical scheme that we used here is of first order in time and second order in space: the truncation error (section 4.1.2) is $O(\Delta t + \Delta x^2)$. It would be possible to use a scheme that is second order in time by using central differences in time too. Alternatively, see exercise 4.10.

4.3.2 Stability analysis

We now analyze the stability of the explicit and implicit schemes for the IBVP in a simple case. The discussion will partly mirror that of sections 4.1.2.1 and 4.1.2.3, but with some complexity added because of the space component.

Let $q \equiv 0$, and assume $T_j^k = \beta^k e^{i\ell x_j}$ for some ℓ[1] . This assumption is intuitively defensible: since the differential equation does not 'mix' the x and t coordinates, we surmise that the solution will be a product $T(x,t) = v(x) \cdot w(t)$ of the separate solutions of

$$\begin{cases} v_{xx} = c_1 v, & v(0) = 0,\ v(1) = 0 \\ w_t = c_2 w & w(0) = 1 \end{cases} \tag{4.71}$$

The only meaningful solution occurs with $c_1, c_2 < 0$, in which case we find:

$$v_{xx} = -c^2 v \Rightarrow v(x) = e^{-icx} = e^{-i\ell\pi x} \tag{4.72}$$

where we substitute $c = \ell\pi$ to take boundary conditions into account, and

$$w(t) = e^{-ct} = e^{-ck\Delta t} = \beta^k, \quad \beta = e^{-ck}. \tag{4.73}$$

If the assumption on this form of the solution holds up, we need $|\beta| < 1$ for stability.

Substituting the surmised form for T_j^k into the explicit scheme gives

$$\begin{aligned} T_j^{k+1} &= T_j^k + \frac{\alpha\Delta t}{\Delta x^2}(T_{j-1}^k - 2T_j^k + T_{j+1}^k) \\ \Rightarrow \beta^{k+1} e^{i\ell x_j} &= \beta^k e^{i\ell x_j} + \frac{\alpha\Delta t}{\Delta x^2}(\beta^k e^{i\ell x_{j-1}} - 2\beta^k e^{i\ell x_j} + \beta^k e^{i\ell x_{j+1}}) \\ &= \beta^k e^{i\ell x_j}\left[1 + \frac{\alpha\Delta t}{\Delta x^2}\left[e^{-i\ell\Delta x} - 2 + e^{i\ell\Delta x}\right]\right] \\ \Rightarrow \beta &= 1 + 2\frac{\alpha\Delta t}{\Delta x^2}[\frac{1}{2}(e^{i\ell\Delta x} + e^{-\ell\Delta x}) - 1] \\ &= 1 + 2\frac{\alpha\Delta t}{\Delta x^2}(\cos(\ell\Delta x) - 1) \end{aligned}$$

For stability we need $|\beta| < 1$:

1. Actually, β is also dependent on ℓ, but we will save ourselves a bunch of subscripts, since different β values never appear together in one formula.

- $\beta < 1 \Leftrightarrow 2\frac{\alpha\Delta t}{\Delta x^2}(\cos(\ell\Delta x) - 1) < 0$: this is true for any ℓ and any choice of $\Delta x, \Delta t$.
- $\beta > -1 \Leftrightarrow 2\frac{\alpha\Delta t}{\Delta x^2}(\cos(\ell\Delta x) - 1) > -2$: this is true for all ℓ only if $2\frac{\alpha\Delta t}{\Delta x^2} < 1$, that is

$$\Delta t < \frac{\Delta x^2}{2\alpha} \tag{4.74}$$

The latter condition poses a big restriction on the allowable size of the time steps: time steps have to be small enough for the method to be stable. This is similar to the stability analysis of the explicit method for the IVP; however, now the time step is also related to the space discretization. This implies that, if we decide we need more accuracy in space and we halve the space discretization Δx, the number of time steps will be multiplied by four.

Let us now consider the stability of the implicit scheme. Substituting the form of the solution $T_j^k = \beta^k e^{i\ell x_j}$ into the numerical scheme gives

$$\begin{aligned} T_j^{k+1} - T_j^k &= \frac{\alpha\Delta t}{\Delta x^2}(T_{j_1}^{k+1} - 2T_j^{k+1} + T_{j+1}^{k+1}) \\ \Rightarrow \beta^{k+1}e^{i\ell\Delta x} - \beta^k e^{i\ell x_j} &= \frac{\alpha\Delta t}{\Delta x^2}(\beta^{k+1}e^{i\ell x_{j-1}} - 2\beta^{k+1}e^{i\ell x_j} + \beta^{k+1}e^{i\ell x_{j+1}}) \end{aligned}$$

Dividing out $e^{i\ell x_j}\beta^{k+1}$ gives

$$\begin{aligned} 1 &= \beta^{-1} + 2\alpha\frac{\Delta t}{\Delta x^2}(\cos\ell\Delta x - 1) \\ \beta &= \frac{1}{1 + 2\alpha\frac{\Delta t}{\Delta x^2}(1 - \cos\ell\Delta x)} \end{aligned}$$

Since $1 - \cos\ell\Delta x \in (0, 2)$, the denominator is strictly > 1. Therefore the condition $|\beta| < 1$ is always satisfied, regardless the value of ℓ and the choices of Δx and Δt: the method is always stable.

Exercise 4.9. Generalize this analysis to two and three space dimensions. Does anything change qualitatively?

Exercise 4.10. The schemes we considered here are of first order in time and second order in space: their discretization order are $O(\Delta t) + O(\Delta x^2)$. Derive the *Crank-Nicolson method* that is obtained by averaging the explicit and implicit schemes, show that it is unconditionally stable, and of second order in time.

Chapter 5

Numerical linear algebra

In chapter 4 you saw how the numerical solution of partial differential equations can lead to linear algebra problems. Sometimes this is a simple problem – a matrix-vector multiplication in the case of the Euler forward method – but sometimes it is more complicated, such as the solution of a system of linear equations in the case of Euler backward methods. Solving linear systems will be the focus of this chapter; in other applications, which we will not discuss here, eigenvalue problems need to be solved.

You may have learned a simple algorithm for solving system of linear equations: elimination of unknowns, also called Gaussian elimination. This method can still be used, but we need some careful discussion of its efficiency. There are also other algorithms, the so-called iterative solution methods, which proceed by gradually approximating the solution of the linear system. They warrant some discussion of their own.

Because of the PDE background, we only consider linear systems that are square and nonsingular. Rectangular, in particular overdetermined, systems have important applications too in a corner of numerical analysis known as optimization theory. However, we will not cover that in this book.

The standard work on numerical linear algebra computations is Golub and Van Loan's *Matrix Computations* [80]. It covers algorithms, error analysis, and computational details. Heath's *Scientific Computing* [97] covers the most common types of computations that arise in scientific computing; this book has many excellent exercises and practical projects.

5.1 Elimination of unknowns

In this section we are going to take a closer look at *Gaussian elimination*, or elimination of unknowns, one of the techniques used to solve a linear system

$$Ax = b$$

for x, given a *coefficient matrix* A and a known right hand side b. You may have seen this method before (and if not, it will be explained below), but we will be a bit more systematic here so that we can analyze various aspects of it.

Remark *It is also possible to solve the equation $Ax = y$ for x by computing the* inverse matrix A^{-1}*, for instance by executing the* Gauss Jordan *algorithm, and multiplying $x \leftarrow A^{-1}x$. The reasons for not doing this are primarily of numerical precision, and fall outside the scope of this book.*

One general thread of this chapter will be the discussion of the efficiency of the various algorithms. If you have already learned in a basic linear algebra course how to solve a system of unknowns by gradually eliminating unknowns, you most likely never applied that method to a matrix larger than 4×4. The linear systems that occur in PDE solving can be thousands of times larger, and counting how many operations they require, as well as how much memory, becomes important.

Let us consider an example of the importance of efficiency in choosing the right algorithm. The solution of a linear system can be written with a fairly simple explicit formula, using determinants. This is called '*Cramer's rule*'. It is mathematically elegant, but completely impractical for our purposes.

If a matrix A and a vector b are given, and a vector x satisfying $Ax = b$ is wanted, then, writing $|A|$ for the determinant,

$$x_i = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1i-1} & b_1 & a_{1i+1} & \dots & a_{1n} \\ a_{21} & & \dots & & b_2 & & \dots & a_{2n} \\ \vdots & & & & \vdots & & & \vdots \\ a_{n1} & & \dots & & b_n & & \dots & a_{nn} \end{vmatrix} / |A|$$

For any matrix M the determinant is defined recursively as

$$|M| = \sum_i (-1)^i m_{1i} |M^{[1,i]}|$$

where $M^{[1,i]}$ denotes the matrix obtained by deleting row 1 and column i from M. This means that computing the determinant of a matrix of dimension n means n times computing a size $n-1$ determinant. Each of these requires $n-1$ determinants of size $n-2$, so you see that the number of operations required to compute the determinant is factorial in the matrix size. This quickly becomes prohibitive, even ignoring any issues of numerical stability. Later in this chapter you will see complexity estimates for other methods of solving systems of linear equations that are considerably more reasonable.

Let us now look at a simple example of solving linear equations with elimination of unknowns. Consider the system

$$\begin{aligned} 6x_1 \quad -2x_2 \quad +2x_3 &= \quad 16 \\ 12x_1 \quad -8x_2 \quad +6x_3 &= \quad 26 \\ 3x_1 \quad -13x_2 \quad +3x_3 &= -19 \end{aligned} \tag{5.1}$$

We eliminate x_1 from the second and third equation by

- multiplying the first equation $\times 2$ and subtracting the result from the second equation, and
- multiplying the first equation $\times 1/2$ and subtracting the result from the third equation.

The linear system then becomes

$$\begin{aligned} 6x_1 \quad -2x_2 \quad +2x_3 &= \quad 16 \\ 0x_1 \quad -4x_2 \quad +2x_3 &= \quad -6 \\ 0x_1 \quad -12x_2 \quad +2x_3 &= -27 \end{aligned}$$

Convince yourself that this system still has the same solution as the original one.

Finally, we eliminate x_2 from the third equation by multiplying the second equation by 3, and subtracting the result from the third equation:

$$\begin{array}{rrrl} 6x_1 & -2x_2 & +2x_3 = & 16 \\ 0x_1 & -4x_2 & +2x_3 = & -6 \\ 0x_1 & +0x_2 & -4x_3 = & -9 \end{array}$$

We can now solve $x_3 = 9/4$ from the last equation. Substituting that in the second equation, we get $-4x_2 = -6-2x_2 = -21/2$ so $x_2 = 21/8$. Finally, from the first equation $6x_1 = 16+2x_2-2x_3 = 16+21/4-9/2 = 76/4$ so $x_1 = 19/6$.

We can write elimination process this more compactly by omitting the x_i coefficients. Write

$$\begin{pmatrix} 6 & -2 & 2 \\ 12 & -8 & 6 \\ 3 & -13 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 16 \\ 26 \\ -19 \end{pmatrix}$$

as

$$\left[\begin{array}{ccc|c} 6 & -2 & 2 & 16 \\ 12 & -8 & 6 & 26 \\ 3 & -13 & 3 & -19 \end{array}\right] \tag{5.2}$$

then the elimination process is

$$\left[\begin{array}{ccc|c} 6 & -2 & 2 & 16 \\ 12 & -8 & 6 & 26 \\ 3 & -13 & 3 & -19 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} 6 & -2 & 2 & 16 \\ 0 & -4 & 2 & -6 \\ 0 & -12 & 2 & -27 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} 6 & -2 & 2 & 16 \\ 0 & -4 & 2 & -6 \\ 0 & 0 & -4 & -9 \end{array}\right].$$

In the above example, the matrix coefficients could have been any real (or, for that matter, complex) coefficients, and you could follow the elimination procedure mechanically. There is the following exception. At some point in the computation, we divided by the numbers $6, -4, -4$ which are found on the diagonal of the matrix in the last elimination step. These quantities are called the *pivots*, and clearly they are required to be nonzero.

Exercise 5.1. The system

$$\begin{array}{rrrl} 6x_1 & -2x_2 & +2x_3 = & 16 \\ 12x_1 & -4x_2 & +6x_3 = & 26 \\ 3x_1 & -13x_2 & +3x_3 = & -19 \end{array}$$

is the same as the one we just investigated in equation (5.1), except for the $(2,2)$ element. Confirm that you get a zero pivot in the second step.

The first pivot is an element of the original matrix. As you saw in the preceding exercise, the other pivots can not be found without doing the actual elimination. In particular, there is no easy way of predicting zero pivots from looking at the system of equations.

If a pivot turns out to be zero, all is not lost for the computation: we can always exchange two matrix rows; this is known as *pivoting*. It is not hard to show (and you can find this in any elementary linear algebra textbook) that with a nonsingular matrix there is always a row exchange possible that puts a nonzero element in the pivot location.

Exercise 5.2. Suppose you want to exchange matrix rows 2 and 3 of the system of equations in equation (5.2). What other adjustments would you have to make to make sure you still compute the correct solution? Continue the system solution of the previous exercise by exchanging rows 2 and 3, and check that you get the correct answer.

Exercise 5.3. Take another look at exercise 5.1. Instead of exchanging rows 2 and 3, exchange columns 2 and 3. What does this mean in terms of the linear system? Continue the process of solving the system; check that you get the same solution as before.

In general, with floating point numbers and round-off, it is very unlikely that a matrix element will become exactly zero during a computation. Also, in a PDE context, the diagonal is usually nonzero. Does that mean that pivoting is in practice almost never necessary? The answer is no: pivoting is desirable from a point of view of numerical stability. In the next section you will see an example that illustrates this fact.

5.2 Linear algebra in computer arithmetic

In most of this chapter, we will act as if all operations can be done in exact arithmetic. However, it is good to become aware of some of the potential problems due to our finite precision computer arithmetic. This allows us to design algorithms that minimize the effect of roundoff. A more rigorous approach to the topic of numerical linear algebra includes a full-fledged error analysis of the algorithms we discuss; however, that is beyond the scope of this course. Error analysis of computations in computer arithmetic is the focus of Wilkinson's classic *Rounding errors in Algebraic Processes* [196] and Higham's more recent *Accuracy and Stability of Numerical Algorithms* [104].

Here, we will only note a few paradigmatic examples of the sort of problems that can come up in computer arithmetic: we will show why pivoting during LU factorization is more than a theoretical device, and we will give two examples of problems in eigenvalue calculations due to the finite precision of computer arithmetic.

5.2.1 Roundoff control during elimination

Above, you saw that row interchanging ('pivoting') is necessary if a zero element appears on the diagonal during elimination of that row and column. Let us now look at what happens if the pivot element is not zero, but close to zero.

Consider the linear system

$$\begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} x = \begin{pmatrix} 1+\epsilon \\ 2 \end{pmatrix}$$

which has the solution solution $x = (1,1)^t$. Using the $(1,1)$ element to clear the remainder of the first column gives:

$$\begin{pmatrix} \epsilon & 1 \\ 0 & 1-\frac{1}{\epsilon} \end{pmatrix} x = \begin{pmatrix} 1+\epsilon \\ 2-\frac{1+\epsilon}{\epsilon} \end{pmatrix} = \begin{pmatrix} 1+\epsilon \\ 1-\frac{1}{\epsilon} \end{pmatrix}.$$

We can now solve x_2 and from it x_1:

$$\begin{cases} x_2 &= (1-\epsilon^{-1})/(1-\epsilon^{-1}) = 1 \\ x_1 &= \epsilon^{-1}(1+\epsilon-x_2) = 1. \end{cases}$$

If ϵ is small, say $\epsilon < \epsilon_{\mathrm{mach}}$, the $1+\epsilon$ term in the right hand side will be 1: our linear system will be

$$\begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} x = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

but the solution $(1,1)^t$ will still satisfy the system in machine arithmetic.

In the first elimination step, $1/\epsilon$ will be very large, so the second component of the right hand side after elimination will be $2 - \frac{1}{\epsilon} = -1/\epsilon$, and the $(2,2)$ element of the matrix is then $-1/\epsilon$ instead of $1 - 1/\epsilon$:

$$\begin{pmatrix} \epsilon & 1 \\ 0 & 1-\epsilon^{-1} \end{pmatrix} x = \begin{pmatrix} 1 \\ 2-\epsilon^{-1} \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} \epsilon & 1 \\ 0 & -\epsilon^{-1} \end{pmatrix} x = \begin{pmatrix} 1 \\ -\epsilon^{-1} \end{pmatrix}$$

Solving first x_2, then x_1, we get:

$$\begin{cases} x_2 &= \epsilon^{-1}/\epsilon^{-1} = 1 \\ x_1 &= \epsilon^{-1}(1-1\cdot x_2) = \epsilon^{-1}\cdot 0 = 0, \end{cases}$$

so x_2 is correct, but x_1 is completely wrong.

Remark *In this example, the numbers of the stated problem are, in computer arithmetic, little off from what they would be in exact arithmetic. Yet the results of the computation can be very much wrong. An analysis of this phenomenon belongs in any good course in numerical analysis. See for instance chapter 1 of [97].*

What would have happened if we had pivoted as described above? We exchange the matrix rows, giving

$$\begin{pmatrix} 1 & 1 \\ \epsilon & 1 \end{pmatrix} x = \begin{pmatrix} 2 \\ 1+\epsilon \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 1 \\ 0 & 1-\epsilon \end{pmatrix} x = \begin{pmatrix} 2 \\ 1-\epsilon \end{pmatrix}$$

Now we get, regardless the size of epsilon:

$$x_2 = \frac{1-\epsilon}{1-\epsilon} = 1, \qquad x_1 = 2 - x_2 = 1$$

In this example we used a very small value of ϵ; a much more refined analysis shows that even with ϵ greater than the machine precision pivoting still makes sense. The general rule of thumb is: *Always do row exchanges to get the largest remaining element in the current column into the pivot position.* In chapter 4 you saw matrices that arise in certain practical applications; it can be shown that for them pivoting is never necessary; see exercise 5.14.

The pivoting that was discussed above is also known as *partial pivoting*, since it is based on row exchanges only. Another option would be *full pivoting*, where row and column exchanges are combined to find the largest element in the remaining subblock, to be used as pivot. Finally, *diagonal pivoting* applies the same exchange to rows and columns. (This is equivalent to renumbering the unknowns of the problem, a strategy which we will consider in section 7.8 for increasing the parallelism of the problem.) This means that pivots are only searched on the diagonal. From now on we will only consider partial pivoting.

5.2.2 Influence of roundoff on eigenvalue computations

Consider the matrix

$$A = \begin{pmatrix} 1 & \epsilon \\ \epsilon & 1 \end{pmatrix}$$

where $\epsilon_{\text{mach}} < |\epsilon| < \sqrt{\epsilon_{\text{mach}}}$, which has eigenvalues $1+\epsilon$ and $1-\epsilon$. If we calculate its characteristic polynomial in computer arithmetic

$$\begin{vmatrix} 1-\lambda & \epsilon \\ \epsilon & 1-\lambda \end{vmatrix} = \lambda^2 - 2\lambda + (1-\epsilon^2) \rightarrow \lambda^2 - 2\lambda + 1.$$

we find a double eigenvalue 1. Note that the exact eigenvalues are expressible in working precision; it is the algorithm that causes the error. Clearly, using the characteristic polynomial is not the right way to compute eigenvalues, even in well-behaved, symmetric positive definite, matrices.

An unsymmetric example: let A be the matrix of size 20

$$A = \begin{pmatrix} 20 & 20 & & & \emptyset \\ & 19 & 20 & & \\ & & \ddots & \ddots & \\ & & & 2 & 20 \\ \emptyset & & & & 1 \end{pmatrix}.$$

Since this is a triangular matrix, its eigenvalues are the diagonal elements. If we perturb this matrix by setting $A_{20,1} = 10^{-6}$ we find a perturbation in the eigenvalues that is much larger than in the elements:

$$\lambda = 20.6 \pm 1.9i, 20.0 \pm 3.8i, 21.2, 16.6 \pm 5.4i, \ldots$$

Also, several of the computed eigenvalues have an imaginary component, which the exact eigenvalues do not have.

5.3 LU factorization

So far, we have looked at eliminating unknowns in the context of solving a single system of linear equations, where we updated the right hand side while reducing the matrix to upper triangular form. Suppose you need to solve more than one system with the same matrix, but with different right hand sides. This happens for instance if you take multiple time steps in an implicit Euler method (section 4.1.2.2). Can you use any of the work you did in the first system to make solving subsequent ones easier?

The answer is yes. You can split the solution process in a part that only concerns the matrix, and a part that is specific to the right hand side. If you have a series of systems to solve, you have to do the first part only once, and, luckily, that even turns out to be the larger part of the work.

Let us take a look at the same example of section 5.1 again.

$$A = \begin{pmatrix} 6 & -2 & 2 \\ 12 & -8 & 6 \\ 3 & -13 & 3 \end{pmatrix}$$

In the elimination process, we took the 2nd row minus 2× the first and the 3rd row minus 1/2× the first. Convince yourself that this combining of rows can be done by multiplying A from the left by

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1/2 & 0 & 1 \end{pmatrix},$$

which is the identity with the elimination coefficients in the first column, below the diagonal. The first step in elimination of variables is equivalent to transforming the system $Ax = b$ to $L_1 Ax = L_1 b$.

Exercise 5.4. Can you find a different L_1 that also has the effect of sweeping the first column? Questions of uniqueness are addressed below in section 5.3.2.

In the next step, you subtracted 3× the second row from the third. Convince yourself that this corresponds to left-multiplying the current matrix $L_1 A$ by

$$L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -3 & 1 \end{pmatrix}$$

We have now transformed our system $Ax = b$ into $L_2 L_1 Ax = L_2 L_1 b$, and $L_2 L_1 A$ is of 'upper triangular' form. If we define $U = L_2 L_1 A$, then $A = L_1^{-1} L_2^{-1} U$. How hard is it to compute matrices such as L_2^{-1}? Remarkably easy, it turns out to be.

We make the following observations:

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1/2 & 0 & 1 \end{pmatrix} \qquad L_1^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1/2 & 0 & 1 \end{pmatrix}$$

and likewise

$$L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -3 & 1 \end{pmatrix} \qquad L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{pmatrix}$$

and even more remarkable:

$$L_1^{-1} L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1/2 & 3 & 1 \end{pmatrix},$$

that is, $L_1^{-1} L_2^{-1}$ contains the off-diagonal elements of L_1^{-1}, L_2^{-1} unchanged, and they in turn contain the elimination coefficients, with only the sign flipped. (This is a special case of *Householder reflectors*; see 14.6.)

Exercise 5.5. Show that a similar statement holds, even if there are elements above the diagonal.

If we define $L = L_1^{-1} L_2^{-1}$, we now have $A = LU$; this is called an *LU factorization*. We see that the coefficients of L below the diagonal are the negative of the coefficients used during elimination. Even better, the first column of L can be written while the first column of A is being eliminated, so the computation of L and U can be done without extra storage, at least if we can afford to lose A.

5.3.1 The LU factorization algorithm

Let us write out the *LU* factorization algorithm in more or less formal code.

⟨*LU* factorization⟩:
    for $k = 1, n-1$:
        ⟨eliminate values in column k⟩
⟨eliminate values in column k⟩:
    for $i = k+1$ to n:
        ⟨compute multiplier for row i⟩
        ⟨update row i⟩
⟨compute multiplier for row i⟩
    $a_{ik} \leftarrow a_{ik}/a_{kk}$
⟨update row i⟩:
    for $j = k+1$ to n:
        $a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

Or, putting everything together in figure 5.1.

⟨*LU* factorization⟩:
    for $k = 1, n-1$:
        for $i = k+1$ to n:
            $a_{ik} \leftarrow a_{ik}/a_{kk}$
            for $j = k+1$ to n:
                $a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

Figure 5.1: LU factorization algorithm

This is the most common way of presenting the LU factorization. However, other ways of computing the same result exist. Algorithms such as the LU factorization can be coded in several ways that are mathematically equivalent, but that have different computational behavior. This issue, in the context of dense matrices, is the focus of van de Geijn and Quintana's *The Science of Programming Matrix Computations* [186].

5.3.1.1 *Computational variants*

The above algorithm is only one of a several ways of computing the $A = LU$ result. That particular formulation was derived by formalizing the process of row elimination. A different algorithm results by looking at the finished result and writing elements a_{ij} in term of elements of L and U.

In particular, let $j \geq k$, then

$$a_{kj} = u_{kj} + \sum_{i<k} \ell_{ki} u_{ij},$$

which gives us the row u_{kj} (for $j \geq k$) expressed in terms of earlier computed U terms:

$$\forall_{j \geq k} : \; u_{kj} = a_{kj} - \sum_{i<k} \ell_{ki} u_{ij}. \tag{5.3}$$

Similarly, with $i > k$,

$$a_{ik} = \ell_{ik}u_{kk} + \sum_{j<k} \ell_{ij}u_{jk}$$

which gives us the column ℓ_{ik} (for $i > k$) as

$$\forall_{i>k} : \ \ell_{ik} = u_{kk}^{-1}(a_{ik} - \sum_{j<k} \ell_{ij}u_{jk}). \tag{5.4}$$

(This is known as *Doolittle's algorithm* for the LU factorization.)

In a classical analysis, this algorithm does the same number of operations, but a computational analysis shows differences. For instance, the prior algorithm would update for each k value the block $i, j > k$, meaning that these elements are repeatedly read and written. This makes it hard to keep these elements in cache.

On the other hand, Doolittle's algorithm computes for each k only one row of U and one column of L. The elements used for this are only read, so they can conceivably be kept in cache.

Another difference appears if we think about parallel execution. Imagine that after computing a pivot in the first algorithm, we compute the next pivot as quickly as possible, and let some other threads do the updating of the $i, j > k$ block. Now we can have potentially several threads all updating the remaining block, which requires synchronization. (This argument will be presented in more detail in section 7.12.)

The new algorithm, on the other hand, has no such conflicting writes: sufficient parallelism can be obtained from the computation of the row of U and column of L.

5.3.1.2 The Cholesky factorization

The LU factorization of a symmetric matrix does not give an L and U that are each other's transpose: L has ones on the diagonal and U has the pivots. However it is possible to make a factorization of a symmetric matrix A of the form $A = LL^t$. This has the advantage that the factorization takes the same amount of space as the original matrix, namely $n(n+1)/2$ elements. We a little luck we can, just as in the LU case, overwrite the matrix with the factorization.

We derive this algorithm by reasoning inductively. Let us write $A = LL^t$ on block form:

$$A = \begin{pmatrix} A_{11} & A_{21}^t \\ A_{21} & A_{22} \end{pmatrix} = LL^t = \begin{pmatrix} \ell_{11} & 0 \\ \ell_{21} & L_{22} \end{pmatrix} \begin{pmatrix} \ell_{11} & \ell_{21}^t \\ 0 & L_{22}^t \end{pmatrix}$$

then $\ell_{11}^2 = a_{11}$, from which we get ℓ_{11}. We also find $\ell_{11}(L^t)_{1j} = \ell_{j1} = a_{1j}$, so we can compute the whole first column of L. Finally, $A_{22} = L_{22}L_{22}^t + \ell_{12}\ell_{12}^t$, so

$$L_{22}L_{22}^t = A_{22} - \ell_{12}\ell_{12}^t,$$

which shows that L_{22} is the Cholesky factor of the updated A_{22} block. Recursively, the algorithm is now defined.

5.3.2 Uniqueness

It is always a good idea, when discussing numerical algorithms, to wonder if different ways of computing lead to the same result. This is referred to as the 'uniqueness' of the result, and it is of practical use: if the computed result is unique, swapping one software library for another will not change anything in the computation.

Let us consider the uniqueness of LU factorization. The definition of an LU factorization algorithm (without pivoting) is that, given a nonsingular matrix A, it will give a lower triangular matrix L and upper triangular matrix U such that $A = LU$. The above algorithm for computing an LU factorization is deterministic (it does not contain instructions 'take any row that satisfies...'), so given the same input, it will always compute the same output. However, other algorithms are possible, so we need to worry whether they give the same result.

Let us then assume that $A = L_1U_1 = L_2U_2$ where L_1, L_2 are lower triangular and U_1, U_2 are upper triangular. Then, $L_2^{-1}L_1 = U_2U_1^{-1}$. In that equation, the left hand side is the product of lower triangular matrices, while the right hand side contains only upper triangular matrices.

Exercise 5.6. Prove that the product of lower triangular matrices is lower triangular, and the product of upper triangular matrices upper triangular. Is a similar statement true for *inverses* of nonsingular triangular matrices?

The product $L_2^{-1}L_1$ is apparently both lower triangular and upper triangular, so it must be diagonal. Let us call it D, then $L_1 = L_2D$ and $U_2 = DU_1$. The conclusion is that LU factorization is not unique, but it *is* unique 'up to diagonal scaling'.

Exercise 5.7. The algorithm in section 5.3.1 resulted in a lower triangular factor L that had ones on the diagonal. Show that this extra condition makes the factorization unique.

Exercise 5.8. Show that an alternative condition of having ones on the diagonal of U is also sufficient for the uniqueness of the factorization.

Since we can demand a unit diagonal in L or in U, you may wonder if it is possible to have both. (Give a simple argument why this is not strictly possible.) We can do the following: suppose that $A = LU$ where L and U are nonsingular lower and upper triangular, but not normalized in any way. Write

$$L = (I + L')D_L, \qquad U = D_U(I + U'), \qquad D = D_LD_U.$$

After some renaming we now have a factorization

$$A = (I + L)D(I + U) \tag{5.5}$$

where D is a diagonal matrix containing the pivots.

Exercise 5.9. Show that you can also normalize the factorization on the form

$$A = (D + L)D^{-1}(D + U). \tag{5.6}$$

How does this D relate to the previous?

Exercise 5.10. Consider the factorization of a tridiagonal matrix on the form 5.6. How do the resulting L and U relate to the triangular parts L_A, U_A of A? Derive a relation between D and D_A and show that this is the equation that generates the pivots.

5.3.3 Pivoting

Above, you saw examples where pivoting, that is, exchanging rows, was necessary during the factorization process, either to guarantee the existence of a nonzero pivot, or for numerical stability. We will now integrate pivoting into the LU factorization.

Let us first observe that row exchanges can be described by a matrix multiplication. Let

$$P^{(i,j)} = \begin{array}{c} \\ i \\ \\ \\ j \\ \\ \end{array} \begin{array}{c} \begin{array}{cccccc} & & & i & & j \end{array} \\ \begin{pmatrix} 1 & 0 & & & & \\ 0 & \ddots & \ddots & & & \\ & & 0 & & 1 & \\ & & & I & & \\ & & 1 & & 0 & \\ & & & & & I \\ & & & & & & \ddots \end{pmatrix} \end{array}$$

then $P^{(i,j)}A$ is the matrix A with rows i and j exchanged. Since we may have to pivot in every iteration i of the factorization process, we introduce a sequence $p(\cdot)$ where $p(i)$ is the j values of the row that row i is switched with. We write $P^{(i)} \equiv P^{(i,p(i))}$ for short.

Exercise 5.11. Show that $P^{(i)}$ is its own inverse.

The process of factoring with partial pivoting can now be described as:

- Let $A^{(i)}$ be the matrix with columns $1 \ldots i-1$ eliminated, and partial pivoting applied to get the desired element in the (i,i) location.
- Let $\ell^{(i)}$ be the vector of multipliers in the i-th elimination step. (That is, the elimination matrix L_i in this step is the identity plus $\ell^{(i)}$ in the i-th column.)
- Let $P^{(i+1)}$ (with $j \geq i+1$) be the matrix that does the partial pivoting for the next elimination step as described above.
- Then $A^{(i+1)} = P^{(i+1)}L_iA^{(i)}$.

In this way we get a factorization of the form

$$L_{n-1}P^{(n-2)}L_{n-2} \cdots L_1P^{(1)}A = U.$$

Suddenly it has become impossible to write $A = LU$: instead we write

$$A = P^{(1)}L_1^{-1} \cdots P^{(n-2)}L_{n-1}^{-1}U. \tag{5.7}$$

Exercise 5.12. Recall from sections 6.10 and 6.8 that blocked algorithms are often desirable from a performance point of view. Why is the 'LU factorization with interleaved pivoting matrices' in equation (5.7) bad news for performance?

Fortunately, equation (5.7) can be simplified: the P and L matrices 'almost commute'. We show this by

looking at an example: $P^{(2)}L_1 = \tilde{L}_1 P^{(2)}$ where $\tilde{L}_1$ is very close to L_1.

$$\begin{pmatrix} 1 & & & \\ & 0 & & 1 \\ & & I & \\ & 1 & & 0 \\ & & & & I \end{pmatrix} \begin{pmatrix} 1 & & \emptyset & \\ \vdots & 1 & & \\ \ell^{(1)} & & \ddots & \\ \vdots & & & 1 \end{pmatrix} = \begin{pmatrix} 1 & & \emptyset & \\ \vdots & 0 & & 1 \\ \tilde{\ell}^{(1)} & & & \\ \vdots & 1 & & 0 \\ \vdots & & & & I \end{pmatrix} = \begin{pmatrix} 1 & & \emptyset & \\ \vdots & 1 & & \\ \tilde{\ell}^{(1)} & & \ddots & \\ \vdots & & & 1 \end{pmatrix} \begin{pmatrix} 1 & & & \\ & 0 & & 1 \\ & & I & \\ & 1 & & 0 \\ & & & & I \end{pmatrix}$$

where $\tilde{\ell}^{(1)}$ is the same as $\ell^{(1)}$, except that elements i and $p(i)$ have been swapped. You can now convince yourself that similarly $P^{(2)}$ et cetera can be 'pulled through' L_1.

As a result we get

$$P^{(n-2)} \cdots P^{(1)} A = \tilde{L}_1^{-1} \cdots L_{n-1}^{-1} U = \tilde{L} U. \tag{5.8}$$

This means that we can again form a matrix L just as before, except that every time we pivot, we need to update the columns of L that have already been computed.

Exercise 5.13. If we write equation (5.8) as $PA = LU$, we get $A = P^{-1}LU$. Can you come up with a simple formula for P^{-1} in terms of just P? Hint: each $P^{(i)}$ is symmetric and its own inverse; see the exercise above.

Exercise 5.14. Earlier, you saw that 2D BVPs (section 4.2.3) give rise to a certain kind of matrix. We stated, without proof, that for these matrices pivoting is not needed. We can now formally prove this, focusing on the crucial property of *diagonal dominance*:

$$\forall_i a_{ii} \geq \sum_{j \neq i} |a_{ij}|.$$

Assume that a matrix A satisfies $\forall_{j \neq i} : \; a_{ij} \leq 0$.

- Show that the matrix is diagonally dominant iff there are vectors $u, v \geq 0$ (meaning that each component is nonnegative) such that $Au = v$.
- Show that, after eliminating a variable, for the remaining matrix $\tilde{A}$ there are again vectors $\tilde{u}, \tilde{v} \geq 0$ such that $\tilde{A}\tilde{u} = \tilde{v}$.
- Now finish the argument that (partial) pivoting is not necessary if A is symmetric and diagonally dominant.

One can actually prove that pivoting is not necessary for any *symmetric positive definite (SPD)* matrix, and diagonal dominance is a stronger condition than SPD-ness.

5.3.4 Solving the system

Now that we have a factorization $A = LU$, we can use this to solve the linear system $Ax = LUx = b$. If we introduce a temporary vector $y = Ux$, we see this takes two steps:

$$Ly = b, \qquad Ux = z.$$

The first part, $Ly = b$ is called the 'lower triangular solve', since it involves the lower triangular matrix L:

$$\begin{pmatrix} 1 & & & & \emptyset \\ \ell_{21} & 1 & & & \\ \ell_{31} & \ell_{32} & 1 & & \\ \vdots & & & \ddots & \\ \ell_{n1} & \ell_{n2} & & \cdots & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \\ \vdots \\ b_n \end{pmatrix}$$

In the first row, you see that $y_1 = b_1$. Then, in the second row $\ell_{21}y_1 + y_2 = b_2$, so $y_2 = b_2 - \ell_{21}y_1$. You can imagine how this continues: in every i-th row you can compute y_i from the previous y-values:

$$y_i = b_i - \sum_{j<i} \ell_{ij} y_j.$$

Since we compute y_i in increasing order, this is also known as the forward substitution, forward solve, or forward sweep.

The second half of the solution process, the upper triangular solve, backward substitution, or backward sweep, computes x from $Ux = y$:

$$\begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ & u_{22} & \dots & u_{2n} \\ & & \ddots & \vdots \\ \emptyset & & & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Now we look at the last line, which immediately tells us that $x_n = u_{nn}^{-1} y_n$. From this, the line before the last states $u_{n-1n-1}x_{n-1} + u_{n-1n}x_n = y_{n-1}$, which gives $x_{n-1} = u_{n-1n-1}^{-1}(y_{n-1} - u_{n-1n}x_n)$. In general, we can compute

$$x_i = u_{ii}^{-1}(y_i - \sum_{j>i} u_{ij} y_j)$$

for decreasing values of i.

Exercise 5.15. In the backward sweep you had to divide by the numbers u_{ii}. That is not possible if any of them are zero. Relate this problem back to the above discussion.

5.3.5 Complexity

In the beginning of this chapter, we indicated that not every method for solving a linear system takes the same number of operations. Let us therefore take a closer look at the complexity (See appendix 15 for a short introduction to complexity.), that is, the number of operations as function of the problem size, of the use of an LU factorization in solving the linear system.

First we look at the computation of x from $LUx = b$ ('solving the linear system'; section 5.3.4), given that we already have the factorization $A = LU$. Looking at the lower and upper triangular part together, you see that you perform a multiplication with all off-diagonal elements (that is, elements ℓ_{ij} or u_{ij} with $i \neq j$). Furthermore, the upper triangular solve involves divisions by the u_{ii} elements. Now, division operations are in general much more expensive than multiplications, so in this case you would compute the values $1/u_{ii}$, and store them instead.

Exercise 5.16. Take a look at the factorization algorithm, and argue that storing the reciprocals of the pivots does not add to the computational complexity.

Summing up, you see that, on a system of size $n \times n$, you perform n^2 multiplications and roughly the same number of additions. This shows that, given a factorization, solving a linear system has the same complexity as a simple matrix-vector multiplication, that is, of computing Ax given A and x.

The complexity of constructing the LU factorization is a bit more involved to compute. Refer to the algorithm in section 5.3.1. You see that in the k-th step two things happen: the computation of the multipliers, and the updating of the rows. There are $n-k$ multipliers to be computed, each of which involve a division. After that, the update takes $(n-k)^2$ additions and multiplications. If we ignore the divisions for now, because there are fewer of them, we find that the LU factorization takes $\sum_{k=1}^{n-1} 2(n-k)^2$ operations. If we number the terms in this sum in the reverse order, we find

$$\#\text{ops} = \sum_{k=1}^{n-1} 2k^2.$$

Since we can approximate a sum by an integral, we find that this is $2/3n^3$ plus some lower order terms. This is of a higher order than solving the linear system: as the system size grows, the cost of constructing the LU factorization completely dominates.

Of course there is more to algorithm analysis than operation counting. While solving the linear system has the same complexity as a matrix-vector multiplication, the two operations are of a very different nature. One big difference is that both the factorization and forward/backward solution involve recursion, so they are not simple to parallelize. We will say more about that later on.

5.3.6 Block algorithms

Many linear algebra matrix operations can be formulated in terms of sub blocks, rather than basic elements. Such blocks sometimes arise naturally from the nature of an application, such as in the case of two-dimensional BVPs; section 4.2.3. However, often we impose a block structure on a matrix purely from a point of performance.

Using *block matrix* algorithms can have several advantages over the traditional scalar view of matrices. For instance, it improves *cache blocking* (section 6.7); it also facilitates scheduling linear algebra algorithms on *multicore* architectures (section 7.12).

For block algorithms we write a matrix as

$$A = \begin{pmatrix} A_{11} & \cdots & A_{1N} \\ \vdots & & \vdots \\ A_{M1} & \cdots & A_{MN} \end{pmatrix}$$

where M, N are the block dimensions, that is, the dimension expressed in terms of the subblocks. Usually, we choose the blocks such that $M = N$ and the diagonal blocks are square.

As a simple example, consider the *matrix-vector product* $y = Ax$, expressed in block terms.

$$\begin{pmatrix} Y_1 \\ \vdots \\ Y_M \end{pmatrix} = \begin{pmatrix} A_{11} & \dots & A_{1M} \\ \vdots & & \vdots \\ A_{M1} & \dots & A_{MM} \end{pmatrix} \begin{pmatrix} X_1 \\ \vdots \\ X_M \end{pmatrix}$$

To see that the block algorithm computes the same result as the old scalar algorithm, we look at a component Y_{i_k}, that is the k-th scalar component of the i-th block. First,

$$Y_i = \sum_j A_{ij} X_j$$

so

$$Y_{i_k} = \big(\sum_j A_{ij} X_j\big)_k = \sum_j \big(A_{ij} X_j\big)_k = \sum_j \sum_\ell A_{i j_{k\ell}} X_{j_\ell}$$

which is the product of the k-th row of the i-th blockrow of A with the whole of X.

A more interesting algorithm is the block version of the *LU factorization*. The algorithm (5.1) then becomes

$$\begin{array}{l} \langle LU \text{ factorization} \rangle: \\ \quad \text{for } k = 1, n-1: \\ \quad\quad \text{for } i = k+1 \text{ to } n: \\ \quad\quad\quad A_{ik} \leftarrow A_{ik} A_{kk}^{-1} \\ \quad\quad\quad \text{for } j = k+1 \text{ to } n: \\ \quad\quad\quad\quad A_{ij} \leftarrow A_{ij} - A_{ik} \cdot A_{kj} \end{array} \tag{5.9}$$

which mostly differs from the earlier algorithm in that the division by a_{kk} has been replaced by a multiplication by A_{kk}^{-1}. Also, the U factor will now have pivot blocks, rather than pivot elements, on the diagonal, so U is only 'block upper triangular', and not strictly upper triangular.

Exercise 5.17. We would like to show that the block algorithm here again computes the same result as the scalar one. Doing so by looking explicitly at the computed elements is cumbersome, so we take another approach. First, recall from section 5.3.2 that LU factorizations are unique if we impose some normalization: if $A = L_1U_1 = L_2U_2$ and L_1, L_2 have unit diagonal, then $L_1 = L_2, U_1 = U_2$.
Next, consider the computation of A_{kk}^{-1}. Show that this can be done by first computing an LU factorization of A_{kk}. Now use this to show that the block LU factorization can give L and U factors that are strictly triangular. The uniqueness of LU factorizations then proves that the block algorithm computes the scalar result.

As a practical point, we note that these matrix blocks are often only conceptual: the matrix is still stored in a traditional row or columnwise manner. The three-parameter `M,N,LDA` description of matrices used in the *BLAS* (section *Tutorials book, section 6*) makes it possible to extract submatrices.

5.4 Sparse matrices

In section 4.2.3 you saw that the discretization of BVPs (and IBVPs) may give rise to sparse matrices. Since such a matrix has N^2 elements but only $O(N)$ nonzeros, it would be a big waste of space to store this as a two-dimensional array. Additionally, we want to avoid operating on zero elements.

In this section we will explore efficient storage schemes for sparse matrices, and the form that familiar linear algebra operations take when using sparse storage.

5.4.1 Storage of sparse matrices

It is pointless to look for an exact definition of *sparse matrix*, but an operational definition is that a matrix is called 'sparse' if there are enough zeros to make specialized storage feasible. We will discuss here briefly the most popular storage schemes for sparse matrices. Since a matrix is no longer stored as a simple 2-dimensional array, algorithms using such storage schemes need to be rewritten too.

5.4.1.1 Band storage and diagonal storage

In section 4.2.2 you have seen examples of sparse matrices that were banded. In fact, their nonzero elements are located precisely on a number of subdiagonals. For such a matrix, a specialized storage scheme is possible.

Let us take as an example the matrix of the one-dimensional BVP (section 4.2.2). Its elements are located on three subdiagonals: the main diagonal and the first super and subdiagonal. In *band storage* we store only the band containing the nonzeros in memory. The most economical storage scheme for such a matrix would store the $2n-2$ elements consecutively. However, for various reasons it is more convenient to waste a few storage locations, as shown in figure 5.2.

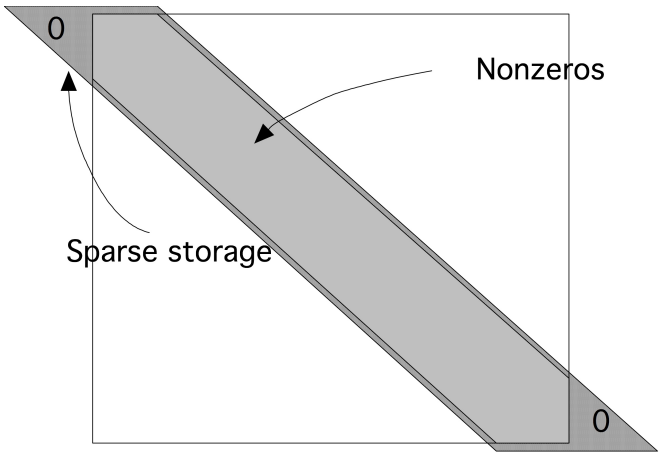


Figure 5.2: Diagonal storage of a banded matrix.

Thus, for a matrix with size $n \times n$ and a *matrix bandwidth* p, we need a rectangular array of size $n \times p$ to

store the matrix. The matrix of equation (4.49) would then be stored as

$$\begin{array}{|c|c|c|}\hline \star & 2 & -1 \\ -1 & 2 & -1 \\ \vdots & \vdots & \vdots \\ -1 & 2 & \star \\ \hline \end{array} \tag{5.10}$$

where the stars indicate array elements that do not correspond to matrix elements: they are the triangles in the top left and bottom right in figure 5.2.

Of course, now we have to wonder about the conversion between array elements `A(i,j)` and matrix elements A_{ij}. We first do this in the Fortran language, with *column-major* storage. If we allocate the array with

```
dimension A(n,-1:1)
```

then the main diagonal A_{ii} is stored in `A(*,0)`. For instance, `A(1,0)` $\sim A_{11}$. The next location in the same row of the matrix A, `A(1,1)` $\sim A_{12}$. It is easy to see that together we have the conversion

$$\texttt{A(i,j)} \sim A_{i,i+j}. \tag{5.11}$$

Remark *The BLAS / Linear Algebra Package (LAPACK) libraries also have a banded storage, but that is column-based, rather than diagonal as we are discussing here.*

Exercise 5.18. What is the reverse conversion, that is, what array location `A(?,?)` does the matrix element A_{ij} correspond to?

Exercise 5.19. If you are a C programmer, derive the conversion between matrix elements A_{ij} and array elements `A[i][j]`.

If we apply this scheme of storing the matrix as an $N \times p$ array to the matrix of the two-dimensional BVP (section 4.2.3), it becomes wasteful, since we would be storing many zeros that exist inside the band. Therefore, in storage by diagonals or *diagonal storage* we refine this scheme by storing only the nonzero diagonals: if the matrix has p nonzero diagonals, we need an $n \times p$ array. For the matrix of equation (4.56) this means:

$$\begin{array}{|ccccc|}\hline \star & \star & 4 & -1 & -1 \\ \vdots & \vdots & 4 & -1 & -1 \\ \vdots & -1 & 4 & -1 & -1 \\ -1 & -1 & 4 & -1 & -1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ -1 & -1 & 4 & \star & \star \\ \hline \end{array}$$

Of course, we need an additional integer array telling us the locations of these nonzero diagonals.

Exercise 5.20. For the central difference matrix in $d = 1, 2, 3$ space dimensions, what is the bandwidth as order of N? What is it as order of the discretization parameter h?

In the preceding examples, the matrices had an equal number of nonzero diagonals above and below the main diagonal. In general this need not be true. For this we introduce the concepts of

- *left halfbandwidth*: if A has a left halfbandwidth of p then $A_{ij} = 0$ for $i > j + p$, and
- *right halfbandwidth*: if A has a right halfbandwidth of p then $A_{ij} = 0$ for $j > i + p$.

If the left and right halfbandwidth are the same, we simply refer to the *halfbandwidth*.

5.4.1.2 Operations on diagonal storage

In several contexts, such as explicit time-stepping (section 4.3.1.1) and iterative solution methods for linear system (section 5.5), the most important operation on sparse matrices is the matrix-vector product. In this section we study how we need to reformulate this operation in view of different storage schemes.

With a matrix stored by diagonals, as described above, it is still possible to perform the ordinary rowwise or columnwise product using the conversion formula (5.11); in fact, this is how Lapack banded routines work. However, with a small bandwidth, this gives short vector lengths and relatively high loop overhead, so it will not be efficient. It is possible do to much better than that.

If we look at how the matrix elements are used in the matrix-vector product, we see that the main diagonal is used as

$$y_i \leftarrow y_i + A_{ii}x_i,$$

the first superdiagonal is used as

$$y_i \leftarrow y_i + A_{ii+1}x_{i+1} \quad \text{for } i < n,$$

and the first subdiagonal as

$$y_i \leftarrow y_i + A_{ii-1}x_{i-1} \quad \text{for } i > 1.$$

In other words, the whole matrix-vector product can be executed in just three vector operations of length n (or $n - 1$), instead of n inner products of length 3 (or 2).

```
for diag = -diag_left, diag_right
    for loc = max(1,1-diag), min(n,n-diag)
        y(loc) = y(loc) + val(loc,diag) * x(loc+diag)
    end
end
```

Exercise 5.21. Write a routine that computes $y \leftarrow A^t x$ by diagonals. Implement it in your favorite language and test it on a random matrix.

Exercise 5.22. The above code fragment is efficient if the matrix is dense inside the band. This is not the case for, for instance, the matrix of two-dimensional BVPs; see section 4.2.3 and in particular equation (4.56). Write code for the matrix-vector product by diagonals that only uses the nonzero diagonals.

Exercise 5.23. Multiplying matrices is harder than multiplying a matrix times a vector. If matrix A has left and halfbandwidth p_A, q_Q, and matrix B has p_B, q_B, what are the left and right halfbandwidth of $C = AB$? Assuming that an array of sufficient size has been allocated for C, write a routine that computes $C \leftarrow AB$.

5.4.1.3 *Compressed row storage*

If we have a sparse matrix that does not have a simple band structure, or where the number of nonzero diagonals becomes impractically large, we use the more general Compressed Row Storage (CRS) scheme. As the name indicates, this scheme is based on compressing all rows, eliminating the zeros; see figure 5.3. Since this loses the information what columns the nonzeros originally came from, we have to store this

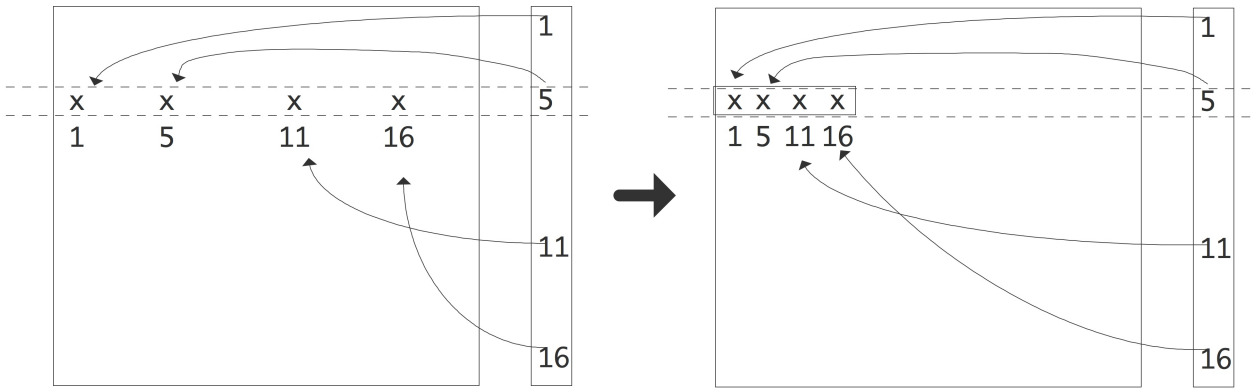


Figure 5.3: Compressing a row of a sparse matrix in the CRS format.

explicitly. Consider an example of a sparse matrix:

$$A = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix}. \tag{5.12}$$

After compressing all rows, we store all nonzeros in a single real array. The column indices are similarly stored in an integer array, and we store pointers to where the columns start. Using 0-based indexing this gives:

`val`	10	−2	3	9	3	7	8	7	3 ··· 9	13	4	2	−1
`colind`	0	4	0	1	5	1	2	3	0 ··· 4	5	1	4	5
`rowptr`	0	2	5	8	12	16	19						

A simple variant of CRS is *Compressed Column Storage (CCS)* where the elements in columns are stored contiguously. This is also known as the *Harwell-Boeing matrix format* [54]. Another storage scheme you may come across is *coordinate storage*, where the matrix is stored as a list of triplets $\langle i, j, a_{ij} \rangle$. The popular *Matrix Market* website [153] uses a variant of this scheme.

5.4.1.4 *Algorithms on compressed row storage*

In this section we will look at the form some algorithms take in CRS.

First we consider the *implementation of the sparse matrix-vector product.*

```
for (row=0; row<nrows; row++) {
   s = 0;
   for (icol=ptr[row]; icol<ptr[row+1]; icol++) {
      int col = ind[icol];
      s += a[icol] * x[col];
   }
   y[row] = s;
}
```

You recognize the standard matrix-vector product algorithm for $y = Ax$, where the inner product is taken of each row A_{i*} and the input vector x. However, the inner loop no long has the column number as index, but rather the location where that number is to be found. This extra step is known as *indirect addressing*.

Exercise 5.24. Compare the data locality of the dense matrix-vector product, executed by rows, with the sparse product given just now. Show that, for general sparse matrices, the spatial locality in addressing the input vector x has now disappeared. Are there matrix structures for which you can still expect some spatial locality?

Now, how about if you wanted to compute the product $y = A^t x$? In that case you need rows of A^t, or, equivalently, columns of A. Finding arbitrary columns of A is hard, requiring lots of searching, so you may think that this algorithm is correspondingly hard to compute. Fortunately, that is not true.

If we exchange the i and j loop in the standard algorithm for $y = Ax$, we get

Original:

$y \leftarrow 0$
for i:
  for j:
    $y_i \leftarrow y_i + a_{ij}x_j$

Indices reversed:

$y \leftarrow 0$
for j:
  for i:
    $y_i \leftarrow y_i + a_{ji}x_j$

We see that in the second variant, columns of A are accessed, rather than rows. This means that we can use the second algorithm for computing the $A^t x$ product by rows.

Exercise 5.25. Write out the code for the transpose product $y = A^t x$ where A is stored in CRS format. Write a simple test program and confirm that your code computes the right thing.

Exercise 5.26. What if you need access to both rows and columns at the same time? Implement an algorithm that tests whether a matrix stored in CRS format is symmetric. Hint: keep an array of pointers, one for each row, that keeps track of how far you have progressed in that row.

Exercise 5.27. The operations described so far are fairly simple, in that they never make changes to the sparsity structure of the matrix. The CRS format, as described above, does not allow you to add new nonzeros to the matrix, but it is not hard to make an extension that does allow it.
Let numbers $p_i, i = 1 \ldots n$, describing the number of nonzeros in the i-th row, be given. Design an extension to CRS that gives each row space for q extra elements. Implement this scheme and test it: construct a matrix with p_i nonzeros in the i-th row, and check the correctness of the matrix-vector product before and after adding new elements, up to q elements per row.
Now assume that the matrix will never have more than a total of qn nonzeros. Alter your code so that it can deal with starting with an empty matrix, and gradually adding nonzeros in random places. Again, check the correctness.

We will revisit the transpose product algorithm in section 7.5.5 in the context of shared memory parallelism.

5.4.1.5 *Ellpack storage variants*

In the above you have seen diagonal storage, which results in long vectors, but is limited to matrices of a definite bandwidth, and compressed row storage, which is more general, but has different vector/parallel properties. Specifically, in CRS, the only vectors are the reduction of a single row times the input vector. These are in general short, and involve indirect addressing, so they will be inefficient for two reasons.

Exercise 5.28. Argue that there is no efficiency problem if we consider the matrix-vector product as a parallel operation.

Vector operations are attractive, first of all for GPUs, but also with the increasing vectorwidth of modern processors; see for instance the instruction set of modern Intel processors. So is there a way to combine the flexibility of CRS and the efficiency of diagonal storage?

The approach taken here is to use a variant of *Ellpack storage*. Here, a matrix is stored by or *jagged diagonals*: the first jagged diagonal consists of the leftmost elements of each row, the second diagonal of the next-leftmost elements, et cetera. Now we can perform a product by diagonals, with the proviso that indirect addressing of the input is needed.

A second problem with this scheme is that the last so many diagonals will not be totally filled. At least two solutions have been proposed here:

- One could sort the rows by rowlength. Since the number of different rowlengths is probably low, this means we will have a short loop over those lengths, around long vectorized loops through the diagonal. On vector architectures this can give enormous performance improvements [39].
- With vector instructions, or on GPUs, only a modest amount of regularity is needed, so one could take a block of 8 or 32 rows, and 'pad' them with zeros.

5.4.2 Sparse matrices and graph theory

Many arguments regarding sparse matrices can be formulated in terms of graph theory. To see why this can be done, consider a matrix A of size n and observe that we can define a graph $\langle E, V \rangle$ by $V = \{1, \ldots, n\}$, $E = \{(i, j) : a_{ij} \neq 0\}$. This is called the *adjacency graph* of the matrix. For simplicity, we assume that A has

a nonzero diagonal. If necessary, we can attach weights to this graph, defined by $w_{ij} = a_{ij}$. The graph is then denoted $\langle E, V, W\rangle$. (If you are not familiar with the basics of graph theory, see appendix 20.)

Graph properties now correspond to matrix properties; for instance, the degree of the graph is the maximum number of nonzeros per row, not counting the diagonal element. As another example, if the graph of the matrix is an *undirected graph*, this means that $a_{ij} \neq 0 \Leftrightarrow a_{ji} \neq 0$. We call such a matrix *structurally symmetric*: it is not truly symmetric in the sense that $\forall_{ij} : a_{ij} = a_{ij}$, but every nonzero in the upper triangle corresponds to one in the lower triangle and vice versa.

5.4.2.1 Graph properties under permutation

One advantage of considering the graph of a matrix is that graph properties do not depend on how we order the nodes, that is, they are invariant under permutation of the matrix.

Exercise 5.29. Let us take a look at what happens with a matrix A when the nodes of its graph $G = \langle V, E, W\rangle$ are renumbered. As a simple example, we number the nodes backwards; that is, with n the number of nodes, we map node i to $n + 1 - i$. Correspondingly, we find a new graph $G' = \langle V, E', W'\rangle$ where

$$(i, j) \in E' \Leftrightarrow (n + 1 - i, n + 1 - j) \in E, \qquad w'_{ij} = w_{n+1-i,n+1-j}.$$

What does this renumbering imply for the matrix A' that corresponds to G'? If you exchange the labels i, j on two nodes, what is the effect on the matrix A?

Exercise 5.30. Some matrix properties stay invariant under permutation. Convince your self that permutation does not change the eigenvalues of a matrix.

Some graph properties can be hard to see from the sparsity pattern of a matrix, but are easier deduced from the graph.

Exercise 5.31. Let A be the tridiagonal matrix of the one-dimensional BVP (see section 4.2.2) of size n with n odd. What does the graph of A look like? Consider the permutation that results from putting the nodes in the following sequence:

$$1, 3, 5, \ldots, n, 2, 4, 6, \ldots, n - 1.$$

What does the sparsity pattern of the permuted matrix look like? Renumbering strategies such as this will be discussed in more detail in section 7.8.2.
Now take this matrix and zero the offdiagonal elements closest to the 'middle' of the matrix: let

$$a_{(n+1)/2,(n+1)/2+1} = a_{(n+1)/2+1,(n+1)/2} = 0.$$

Describe what that does to the graph of A. Such a graph is called *reducible*. Now apply the permutation of the previous exercise and sketch the resulting sparsity pattern. Note that the reducibility of the graph is now harder to read from the sparsity pattern.

5.4.3 LU factorizations of sparse matrices

In section 4.2.2 the one-dimensional BVP led to a linear system with a tridiagonal coefficient matrix. If we do one step of Gaussian elimination, the only element that needs to be eliminated is in the second row:

$$\begin{pmatrix} 2 & -1 & 0 & \dots & \\ -1 & 2 & -1 & & \\ 0 & -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots & \ddots \end{pmatrix} \Rightarrow \left(\begin{array}{c|ccc} 2 & -1 & 0 & \dots \\ \hline 0 & 2-\frac{1}{2} & -1 & \\ 0 & -1 & 2 & -1 \\ & \ddots & \ddots & \ddots & \ddots \end{array}\right)$$

There are two important observations to be made: one is that this elimination step does not change any zero elements to nonzero. The other observation is that the part of the matrix that is left to be eliminated is again tridiagonal. Inductively, during the elimination no zero elements change to nonzero: the sparsity pattern of $L+U$ is the same as of A, and so the factorization takes the same amount of space to store as the matrix.

The case of tridiagonal matrices is unfortunately not typical, as we will shortly see in the case of two-dimensional problems. But first we will extend the discussion on graph theory of section 5.4.2 to factorizations.

5.4.3.1 *Graph theory of sparse LU factorization*

Graph theory is often useful when discussion the *LU factorization of a sparse matrix*. Let us investigate what eliminating the first unknown (or sweeping the first column) means in graph theoretic terms. We are assuming a structurally symmetric matrix.

We consider eliminating an unknown as a process that takes a graph $G = \langle V, E\rangle$ and turns it into a graph $G' = \langle V', E'\rangle$. The relation between these graphs is first that a vertex, say k, has been removed from the vertices: $k \notin V', V' \cup \{k\} = V$.

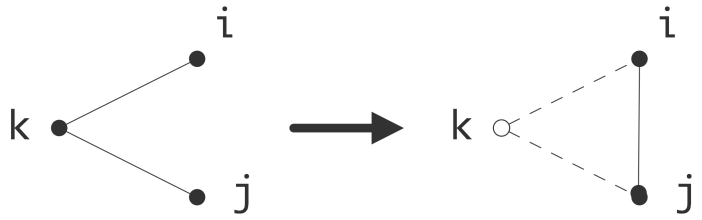


Figure 5.4: Eliminating a vertex introduces a new edge in the quotient graph.

The relationship between E and E' is more complicated. In the Gaussian elimination algorithm the result of eliminating variable k is that the statement

$$a_{ij} \leftarrow a_{ij} - a_{ik}a_{kk}^{-1}a_{kj}$$

is executed for all $i,j \neq k$. If $a_{ij} \neq 0$ originally, that is, $(i,j) \in E$, then the value of a_{ij} is merely altered, which does not change the adjacency graph. In case $a_{ij} = 0$ in the original matrix, meaning $(i,j) \notin E$, there will be a nonzero element, termed a *fill-in* element, after the k unknown is eliminated [162]:

$$(i,j) \notin E \quad \text{but} \quad (i,j) \in E'.$$

This is illustrated in figure 5.4.

Summarizing, eliminating an unknown gives a graph that has one vertex less, and that has additional edges for all i, j such that there were edges between i or j and the eliminated variable k.

Figure 5.5 gives a full illustration on a small matrix.

Exercise 5.32. Go back to exercise 5.31. Use a graph argument to determine the sparsity pattern after the odd variables have been eliminated.

Exercise 5.33. Prove the generalization of the above argument about eliminating a single vertex. Let $I \subset V$ be any set of vertices, and let J be the vertices connected to I:

$$J \cap I = \emptyset, \quad \forall_{i \in I} \exists_{j \in J} : (i,j) \in E.$$

Now show that eliminating the variables in I leads to a graph $\langle V', E' \rangle$ where all nodes in J are connected in the remaining graph, if there was a path between them through I:

$$\forall_{j_1, j_2 \in J} : \text{ there is a path } j_i \rightarrow j_2 \text{ through } I \text{ in } E \Rightarrow (j_1, j_2) \in E'.$$

5.4.3.2 *Fill-in*

We now return to the factorization of the matrix from two-dimensional problems. We write such matrices of size $N \times N$ as block matrices with block dimension n, each block being of size n. (Refresher question: where do these blocks come from?) Now, in the first elimination step we need to zero two elements, a_{21} and $a_{n+1,1}$.

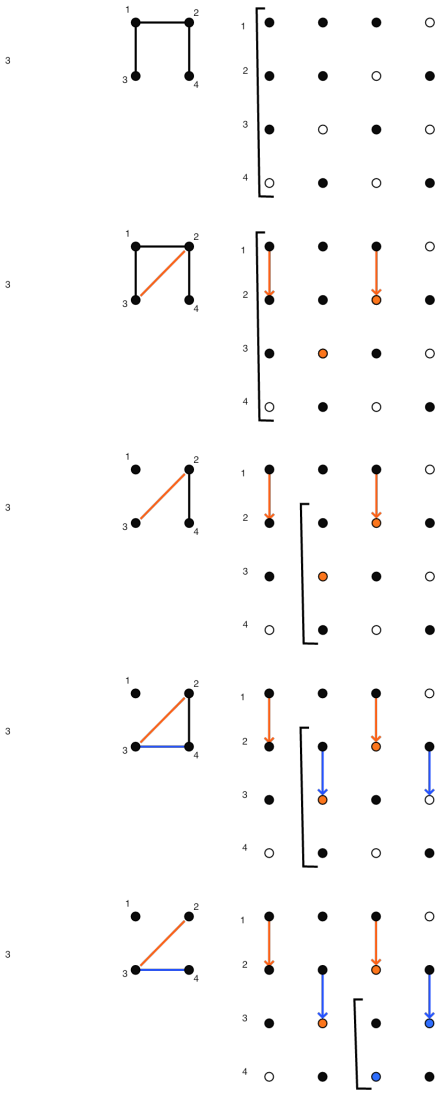
$$\left(\begin{array}{ccccc:cc:c}
4 & -1 & 0 & \dots & & -1 & & \\
-1 & 4 & -1 & 0 & \dots & 0 & -1 & \\
 & \ddots & \ddots & \ddots & & & \ddots & \\
\hdashline
-1 & 0 & \dots & & & 4 & -1 & \\
0 & -1 & 0 & \dots & & -1 & 4 & -1
\end{array}\right) \Rightarrow \left(\begin{array}{c:ccccc:cc:c}
4 & -1 & 0 & \dots & & -1 & & \\
\hdashline
 & 4-\frac{1}{4} & -1 & 0 & \dots & -1/4 & -1 & \\
 & \ddots & \ddots & \ddots & & & \ddots & \ddots \\
\hdashline
 & -1/4 & & & & 4-\frac{1}{4} & -1 & \\
 & -1 & 0 & & & -1 & 4 & -1
\end{array}\right)$$

You see that eliminating a_{21} and $a_{n+1,1}$ causes two *fill* elements to appear: in the original matrix $a_{2,n+1}$ and $a_{n+1,2}$ are zero, but in the modified matrix these locations are nonzero. We define *fill locations* as locations (i,j) where $a_{ij} = 0$, but $(L+U)_{ij} \neq 0$.

Clearly the matrix fills in during factorization. With a little imagination you can also see that every element in the band outside the first diagonal block will fill in. However, using the graph approach of section 5.4.3.1 it becomes easy to visualize the fill-in connections that are created.

In figure 5.6 this is illustrated for the graph of the 2d BVP example. (The edges corresponding to diagonal elements have not been pictured here.) Each variable in the first row that is eliminated creates connections between the next variable and the second row, and between variables in the second row. Inductively you see that after the first row is eliminated the second row is fully connected. (Connect this to exercise 5.33.)

Exercise 5.34. Finish the argument. What does the fact that variables in the second row are fully connected imply for the matrix structure? Sketch in a figure what happens after the first variable in the second row is eliminated.



Original matrix.

Eliminating $(2,1)$ causes fill-in at $(2,3)$.

Remaining matrix when step 1 finished.

Eliminating $(3,2)$ fills $(3,4)$

After step 2

Figure 5.5: Gaussian elimination of a sparse matrix.

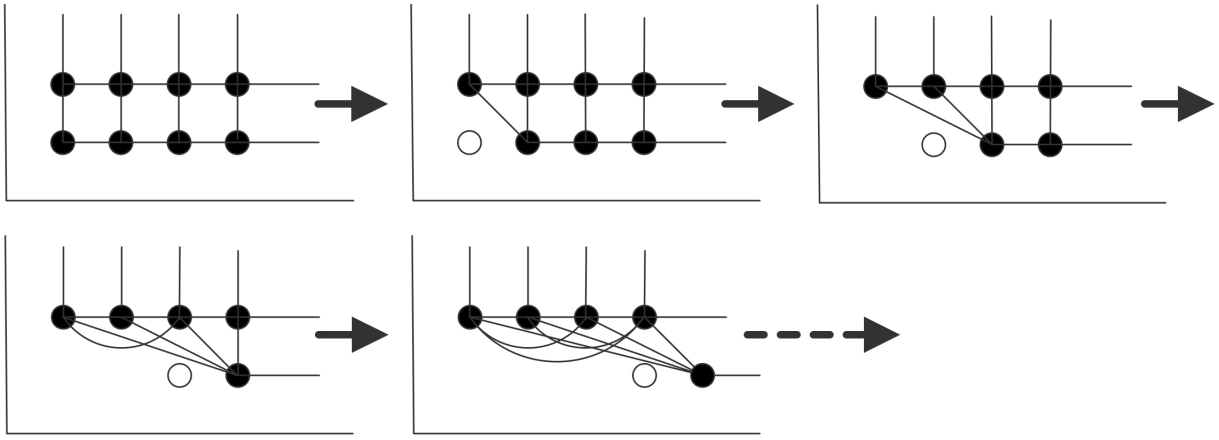


Figure 5.6: Creation of fill-in connection in the matrix graph.

Exercise 5.35. The LAPACK software for dense linear algebra has an LU factorization routine that overwrites the input matrix with the factors. Above you saw that is possible since the columns of L are generated precisely as the columns of A are eliminated. Why is such an algorithm not possible if the matrix is stored in sparse format?

5.4.3.3 *Fill-in estimates*

In the above example you saw that the factorization of a sparse matrix can take much more space than the matrix itself, but still less than storing an entire square array of size the matrix dimension. We will now give some bounds for the *space complexity* of the factorization, that is, the amount of space needed to execute the factorization algorithm.

Exercise 5.36. Prove the following statements.

1. Assume that the matrix A has a *halfbandwidth* p, that is, $a_{ij} = 0$ if $|i-j| > p$. Show that, after a factorization without pivoting, $L + U$ has the same halfbandwidth.
2. Show that, after a factorization with partial pivoting, L has a left halfbandwidth of p, whereas U has a right halfbandwidth of $2p$.
3. Assuming no pivoting, show that the fill-in can be characterized as follows:

   Consider row i. Let $j_{\min}$ be the leftmost nonzero in row i, that is $a_{ij} = 0$ for $j < j_{\min}$. Then there will be no fill-in in row i to the left of column $j_{\min}$. Likewise, if $i_{\min}$ is the topmost nonzero in column j, there will be no fill-in in column j above row $i_{\min}$.

   As a result, L and U have a 'skyline' profile. Given a sparse matrix, it is now easy to allocate enough storage to fit a factorization without pivoting: this is knows as *skyline storage*.

Exercise 5.37. Consider the matrix

$$A = \begin{pmatrix} a_{11} & 0 & a_{13} & & & & & \emptyset \\ 0 & a_{22} & 0 & a_{24} & & & & \\ a_{31} & 0 & a_{33} & 0 & a_{35} & & & \\ & a_{42} & 0 & a_{44} & 0 & a_{46} & & \\ & & \ddots & \ddots & \ddots & \ddots & \ddots & \\ \emptyset & & & & & a_{n,n-1} & 0 & a_{nn} \end{pmatrix}$$

You have proved earlier that any fill-in from performing an LU factorization is limited to the band that contains the original matrix elements. In this case there is no fill-in. Prove this inductively.
Look at the adjacency graph. (This sort of graph has a name. What is it?) Can you give a proof based on this graph that there will be no fill-in?

Exercise 5.36 shows that we can allocate enough storage for the factorization of a banded matrix:

- for the factorization without pivoting of a matrix with bandwidth p, an array of size $N \times p$ suffices;
- the factorization with partial pivoting of a matrix left halfbandwidth p and right halfbandwidth q can be stored in $N \times (p + 2q + 1)$.
- A skyline profile, sufficient for storing the factorization, can be constructed based on the specific matrix.

We can apply this estimate to the matrix from the two-dimensional BVP, section 4.2.3.

Exercise 5.38. Show that in equation (4.56) the original matrix has $O(N) = O(n^2)$ nonzero elements, $O(N^2) = O(n^4)$ elements in total, and the factorization has $O(nN) = O(n^3) = O(N^{3/2})$ nonzeros.

These estimates show that the storage required for an LU factorization can be more than what is required for A, and the difference is not a constant factor, but related to the matrix size. Without proof we state that the inverses of the kind of sparse matrices you have seen so far are fully dense, so storing them takes even more. This is an important reason that solving linear systems $Ax = y$ is not done in practice by computing A^{-1} and subsequently multiplying $x = A^{-1}y$. (Numerical stability is another reason that this is not done.) The fact that even a factorization can take a lot of space is one reason for considering iterative methods, as we will do in section 5.5.

Above, you saw that the factorization of a dense matrix of size $n \times n$ takes $O(n^3)$ operations. How is this for a sparse matrix? Let us consider the case of a matrix with halfbandwidth p, and assume that the original matrix is dense in that band. The pivot element a_{11} is used to zero p elements in the first column, and for each the first row is added to that row, involving p multiplications and additions. In sum, we find that the number of operations is roughly

$$\sum_{i=1}^{n} p^2 = p^2 \cdot n$$

plus or minus lower order terms.

Exercise 5.39. The assumption of a band that is initially dense is not true for the matrix of a two-dimensional BVP. Why does the above estimate still hold, up to some lower order terms?

In exercise 5.36 above you derived an estimate for the amount of fill-in that is easy to apply. However, it can be a considerable overestimate. It is desirable to compute or estimate the amount of fill-in with less work than doing the actual factorization. We will now sketch an algorithm for finding the exact number of nonzeros in $L+U$, with a cost that is linear in this number. We will do this in the (structurally) symmetric case. The crucial observation is the following. Suppose column i has more than one nonzero below the diagonal:

$$\begin{pmatrix} \ddots & & & & \\ & a_{ii} & & a_{ij} & & a_{ik} \\ & & \ddots & & & \\ & a_{ji} & & a_{jj} & & \\ & & & & \ddots & \\ & a_{ki} & & ?a_{kj}? & & a_{kk} \end{pmatrix}$$

Eliminating a_{ki} in the i-th step causes an update of a_{kj}, or a fill-in element if originally $a_{kj} = 0$. However, we can infer the existence of this nonzero value: eliminating a_{ji} causes a fill-in element in location (j,k), and we know that structural symmetry is preserved. In other words, if we are only counting nonzeros, it is enough to look at the effects of eliminating the (j,i) location, or in general the first nonzero below the pivot. Following this argument through, we only need to record the nonzeros in one row per pivot, and the entire process has a complexity linear in the number of nonzeros in the factorization.

5.4.3.4 *Fill-in reduction*

Graph properties of a matrix, such as degree and diameter, are invariant under renumbering the variables. Other properties, such as fill-in during a factorization, are affected by renumbering. In fact, it is worthwhile investigating whether it is possible to reduce the amount of fill-in by renumbering the nodes of the matrix graph, or equivalently, by applying a *permutation* to the linear system.

Exercise 5.40. Consider the 'arrow' matrix with nonzeros only in the first row and column and on the diagonal:

$$\begin{pmatrix} * & * & \cdots & * \\ * & * & & \emptyset \\ \vdots & & \ddots & \\ * & \emptyset & & * \end{pmatrix}$$

What is the number of nonzeros in the matrix, and in the factorization, assuming that no addition ever results in zero? Can you find a symmetric permutation of the variables of the problem such that the new matrix has no fill-in?

This example is not typical, but it is true that fill-in estimates can sometimes be improved upon by clever permuting of the matrix (see for instance section 7.8.1). Even with this, as a rule the statement holds that an LU factorization of a sparse matrix takes considerably more space than the matrix itself. This is one of the motivating factors for the iterative methods in the next section.

5.4.3.5 *Fill-in reducing orderings*

Some matrix properties are invariant under symmetric permutations.

Exercise 5.41. In linear algebra classes, you typically look at matrix properties and whether they are invariant under a change of basis, in particular under *unitary basis transformations*:

$$B = VAV^t, \quad \text{where } VV^t = I.$$

Show that a symmetric permutation is a particular change of basis. Name some matrix properties that do not change under unitary transformations.

Other properties are not: in the previous section you saw that the amount of fill-in is one of those. Thus, you may wonder what the best ordering is to reduce the fill-in of factoring a given matrix. This problem is intractable in practice, but various heuristics exist. Some of these heuristics can also be justified from a point of view of parallelism; in fact, the *nested dissection* ordering will only be discussed in the section on parallelism 7.8.1. Here we briefly show two other heuristics that predate the need for parallelism.

First we will look at the *Cuthill-McKee ordering* which directly minimizes the bandwidth of the permuted matrix. Since the amount of fill-in can be bounded in terms of the bandwidth, we hope that such a bandwidth reducing ordering will also reduce the fill-in.

Secondly, we will consider the *minimum degree ordering*, which aims more directly for fill-in reduction.

5.4.3.5.1 **Cuthill-McKee ordering**

The *Cuthill-McKee ordering* [37] is a *bandwidth reducing ordering* that works by ordering the variables in *level sets*; figure 5.7. It considers the *adjacency graph* of the matrix, and proceeds as follows:

1. Take an arbitrary node, and call that 'level zero'.
2. Given a level n, assign all nodes that are connecting to level n, and that are not yet in a level, to level $n + 1$.
3. For the so-called 'reverse Cuthill-McKee ordering', reverse the numbering of the levels.

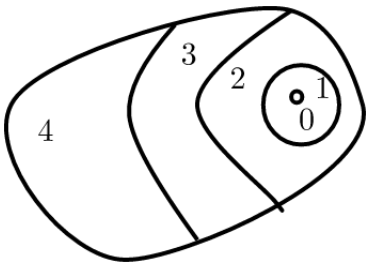


Figure 5.7: Level sets.

Exercise 5.42. Show that permuting a matrix according to the Cuthill-McKee ordering has a *block tridiagonal* structure.

We will revisit this algorithm in section 7.10.1 when we consider parallelism.

Of course, one can wonder just how far the bandwidth can be reduced.

Exercise 5.43. The *diameter* of a graph is defined as the maximum shortest distance between two nodes.

1. Argue that in the graph of a 2D elliptic problem this diameter is $O(N^{1/2})$.
2. Express the path length between nodes 1 and N in terms of the bandwidth.
3. Argue that this gives a lower bound on the diameter, and use this to derive a lower bound on the bandwidth.

5.4.3.5.2 **Minimum degree ordering** Another ordering is motivated by the observation that the amount of fill-in is related to the degree of nodes.

Exercise 5.44. Show that eliminating a node with degree d leads to at most $2d$ fill elements

The so-called *minimum degree ordering* proceeds as follows:

- Find the node with lowest degree;
- eliminate that node and update the degree information for the remaining nodes;
- repeat from the first step, with the updated matrix graph.

Exercise 5.45. Indicate a difference between the two above-mentioned methods. Both are based on inspection of the matrix graph; however, the minimum degree method requires much more flexibility in the data structures used. Explain why and discuss two aspects in detail.

5.5 Iterative methods

Gaussian elimination, the use of an LU factorization, is a simple way to find the solution of a linear system, but as we saw above, in the sort of problems that come from discretized PDEs, it can create a lot of fill-in. In this section we will look at a completely different approach, *iterative solution*, where the solution of the system is found by a sequence of approximations.

The computational scheme looks, very roughly, like:

$$\begin{cases} \text{Choose any starting vector } x_0 \text{ and repeat for } i \geq 0: \\ x_{i+1} = Bx_i + c \\ \text{until some stopping test is satisfied.} \end{cases}$$

The important feature here is that no systems are solved with the original coefficient matrix; instead, every iteration involves a matrix-vector multiplication or a solution of a much simpler system. Thus we have replaced a complicated operation, constructing an LU factorization and solving a system with it, by a repeated simpler and cheaper operation. This makes iterative methods easier to code, and potentially more efficient.

Let us consider a simple example to motivate the precise definition of the iterative methods. Suppose we want to solve the system

$$\begin{pmatrix} 10 & 0 & 1 \\ 1/2 & 7 & 1 \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

which has the solution $(2, 1, 1)$. Suppose you know (for example, from physical considerations) that solution components are roughly the same size. Observe the dominant size of the diagonal, then, to decide that

$$\begin{pmatrix} 10 & & \\ & 7 & \\ & & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

might be a good approximation. This has the solution $(2.1, 9/7, 8/6)$. Clearly, solving a system that only involves the diagonal of the original system is both easy to do, and, at least in this case, fairly accurate.

Another approximation to the original system would be to use the lower triangle. The system

$$\begin{pmatrix} 10 & & \\ 1/2 & 7 & \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

has the solution $(2.1, 7.95/7, 5.9/6)$. Solving triangular systems is a bit more work than diagonal systems, but still a lot easier than computing an LU factorization. Also, we have not generated any fill-in in the process of finding this approximate solution.

Thus we see that there are easy to compute ways of getting reasonably close to the solution. Can we somehow repeat this trick?

Formulated a bit more abstractly, what we did was instead of solving $Ax = b$ we solved $L\tilde{x} = b$. Now define Δx as the distance from the true solution: $\tilde{x} = x + \Delta x$. This gives $A\Delta x = A\tilde{x} - b \equiv r$, where r is the *residual*. Next we solve again $L\widetilde{\Delta x} = r$ and update $\tilde{\tilde{x}} = \tilde{x} - \widetilde{\Delta x}$.

iteration	1	2	3
x_1	2.1000	2.0017	2.000028
x_2	1.1357	1.0023	1.000038
x_3	0.9833	0.9997	0.999995

In this case we get two decimals per iteration, which is not typical.

It is now clear why iterative methods can be attractive. Solving a system by Gaussian elimination takes $O(n^3)$ operations, as shown above. A single iteration in a scheme as the above takes $O(n^2)$ operations if the matrix is dense, and possibly as low as $O(n)$ for a sparse matrix. If the number of iterations is low, this makes iterative methods competitive.

Exercise 5.46. When comparing iterative and direct methods, the flop count is not the only relevant measure. Outline some issues relating to the efficiency of the code in both cases. Also compare the cases of solving a single linear system and solving multiple.

5.5.1 Abstract presentation

It is time to do a formal presentation of the iterative scheme of the above example. Suppose we want to solve $Ax = b$, and a direct solution is too expensive, but multiplying by A is feasible. Suppose furthermore that we have a matrix $K \approx A$ such that solving $Kx = b$ can be done cheaply.

Instead of solving $Ax = b$ we solve $Kx = b$, and define x_0 as the solution: $Kx_0 = b$. This leaves us with an error $e_0 = x_0 - x$, for which we have the equation $A(x_0 - e_0) = b$ or $Ae_0 = Ax_0 - b$. We call $r_0 \equiv Ax_0 - b$ the *residual*; the error then satisfies $Ae_0 = r_0$.

If we could solve the error from the equation $Ae_0 = r_0$, we would be done: the true solution is then found as $x = x_0 - e_0$. However, since solving with A was too expensive the first time, we can not do so this time

either, so we determine the error correction approximately. We solve $K\tilde{e}_0 = r_0$ and set $x_1 := x_0 - \tilde{e}_0$; the story can now continue with $e_1 = x_1 - x$, $r_1 = Ax_1 - b$, $K\tilde{e}_1 = r_1$, $x_2 = x_1 - \tilde{e}_1$, et cetera.

The iteration scheme is then:

Let x_0 be given
For $i \geq 0$:
    let $r_i = Ax_i - b$
    compute e_i from $Ke_i = r_i$
    update $x_{i+1} = x_i - e_i$

We call the basic scheme

$$x_{i+1} = x_i - K^{-1}r_i \tag{5.13}$$

a *stationary iteration*. It is stationary because every update is performed the same way, without any dependence on the iteration number. This scheme has a simple analysis, but unfortunately limited applicability.

There are several questions we need to answer about iterative schemes:

- Does this scheme always take us to the solution?
- If the scheme converges, how quickly?
- When do we stop iterating?
- How do we choose K?

We will now devote some attention to these matters, though a full discussion is beyond the scope of this book.

5.5.2 Convergence and error analysis

We start with the question of whether the iterative scheme converges, and how quickly. Consider one iteration step:

$$\begin{aligned} r_1 &= Ax_1 - b = A(x_0 - \tilde{e}_0) - b \\ &= r_0 - AK^{-1}r_0 \\ &= (I - AK^{-1})r_0 \end{aligned} \tag{5.14}$$

Inductively we find $r_n = (I - AK^{-1})^n r_0$, so $r_n \downarrow 0$ if all eigenvalues satisfy $|\lambda(I - AK^{-1})| < 1$[1].

This last statement gives us both a condition for convergence, by relating K to A, and a geometric convergence rate, if K is close enough.

Exercise 5.47. Derive a similar inductive relation for e_n.

It is hard to determine if the condition $|\lambda(I - AK^{-1})| < 1$ is satisfied by computing the actual eigenvalues. However, sometimes the *Gershgorin theorem* (appendix 14.5) gives us enough information.

Exercise 5.48. Consider the matrix A of equation (4.56) that we obtained from discretization of a two-dimensional BVP. Let K be matrix containing the diagonal of A, that is $k_{ii} = a_{ii}$ and $k_{ij} = 0$ for $i \neq j$. Use the Gershgorin theorem to show that $|\lambda(I - AK^{-1})| < 1$.

1. This is fairly easy to see in the case where the matrix is diagonalizable and has a full basis of eigenvectors. However, it is true in the general case too.

The argument in this exercise is hard to generalize for more complicated choices of K, such as you will see below. Here we only remark that for certain matrices A, these choices of K will always lead to convergence, with a speed that decreases as the matrix size increases. We will not go into the details beyond stating that for M-matrices (see section 4.2.2) these iterative methods converge. For more details on the convergence theory of stationary iterative methods, see [190]

5.5.3 Computational form

Above, in section 5.5.1, we derived stationary iteration as a process that involves multiplying by A and solving with K. However, in some cases a simpler implementation is possible. Consider the case where $A = K - N$, and we know both K and N. Then we write $Ax = b$ as

$$Kx = Nx + b \tag{5.15}$$

and we observe that an x satisfying (5.15) is a fixed point of the iteration

$$Kx^{(n+1)} = Nx^{(i)} + b.$$

It is easy to see that this is a stationary iteration:

$$\begin{aligned} Kx^{(n+1)} &= Nx^{(i)} + b \\ &= Kx^{(n)} - Ax^{(n)} + b \\ &= Kx^{(n)} - r^{(n)} \\ \Rightarrow x^{(n+1)} &= x^{(n)} - K^{-1}r^{(n)}. \end{aligned}$$

which is the basic form of equation (5.13). The convergence criterion $|\lambda(I - AK^{-1})| < 1$ (see above) now simplifies to $|\lambda(NK^{-1})| < 1$.

Let us consider some special cases. First of all, let $K = D_A$, that is, the matrix containing the diagonal part of A: $k_{ii} = a_{ii}$ and $k_{ij} = 0$ for all $i \neq j$. Likewise, $n_{ii} = 0$ and $n_{ij} = -a_{ij}$ for all $i \neq j$.

This is known as the *Jacobi iteration*. The matrix formulation $Kx^{(n+1)} = Nx^{(n)} + b$ can be written pointwise as

$$\forall_i : a_{ii}x_i^{(t+1)} = \sum_{j \neq i} a_{ij}x_j^{(t)} + b_i \tag{5.16}$$

which becomes

for $t = 1, \ldots$ until convergence, do:
  for $i = 1 \ldots n$:
    $x_i^{(t+1)} = a_{ii}^{-1}(\sum_{j \neq i} a_{ij}x_j^{(t)} + b_i)$

(Bearing in mind that divisions are relatively costly, see section 1.2, we would actually store the a_{ii}^{-1} quantities explicitly, and replace the division by a multiplication.)

This requires us to have one vector x for the current iterate $x^{(t)}$, and one temporary u for the next vector $x^{(t+1)}$. The easiest way to write this is:

for $t = 1, \ldots$ until convergence, do:
  for $i = 1 \ldots n$:
    $u_i = a_{ii}^{-1}(-\sum_{j \neq i} a_{ij} x_j + b_i)$
  copy $x \leftarrow u$

For the simple case of a one-dimensional problem this is illustrated in figure 5.8: in each x_i point the values

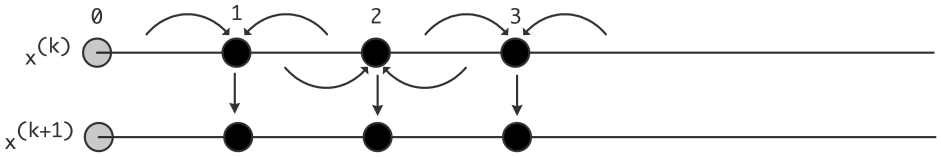


Figure 5.8: Data movement pattern in the Jacobi iteration on a one-dimensional problem.

of the two neighbors are combined with the current value to generate a new value. Since the computations in all the x_i points are independent, this can be done in parallel on a parallel computer.

But, you might think, in the sum $\sum_{j \neq i} a_{ij} x_j$ why not use the $x^{(t+1)}$ values for as far as already computed? In terms of the vectors $x^{(t)}$ this means

for $k = 1, \ldots$ until convergence, do:
  for $i = 1 \ldots n$:
    $x_i^{(t+1)} = a_{ii}^{-1}(-\sum_{j<i} a_{ij} x_j^{(t+1)} - \sum_{j>i} a_{ij} x_j^{(t)} + b_i)$

Surprisingly, the implementation is simpler than of the Jacobi method:

for $t = 1, \ldots$ until convergence, do:
  for $i = 1 \ldots n$:
    $x_i = a_{ii}^{-1}(-\sum_{j \neq i} a_{ij} x_j + b_i)$

If you write this out as a matrix equation, you see that the newly computed elements elements $x_i^{(t+1)}$ are multiplied with elements of $D_A + L_A$, and the old elements $x_j^{(t)}$ by U_A, giving

$$(D_A + L_A)x^{(k+1)} = -U_A x^{(k)} + b$$

which is called the *Gauss-Seidel* method.

For the one-dimensional case, the Gauss-Seidel method is illustrated in figure 5.9; every x_i point still combines its neighbors' values, but now the left value is actually from the next outer iteration.

Finally, we can insert a damping parameter into the Gauss-Seidel scheme, giving the *Successive Over-Relaxation (SOR)* method:

for $t = 1, \ldots$ until convergence, do:
  for $i = 1 \ldots n$:
    $x_i^{(t+1)} = \omega a_{ii}^{-1}(-\sum_{j<i} a_{ij} x_j^{(t+1)} - \sum_{j>i} a_{ij} x_j^{(t)} + b_i) + (1 - \omega)x^{(t)}$

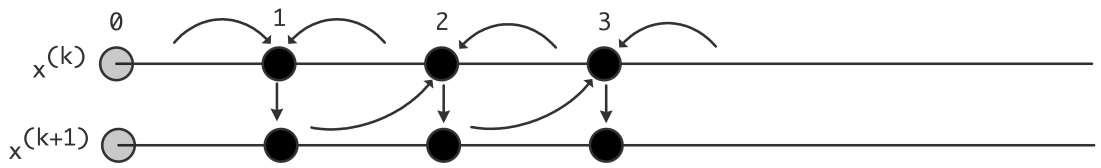


Figure 5.9: Data movement pattern in the Gauss-Seidel iteration on a one-dimensional problem.

Surprisingly for something that looks like an interpolation, the method actually works with values for ω in the range $\omega \in (0, 2)$, the optimal value being larger than 1 [94]. Computing the optimal ω is not simple.

5.5.4 Convergence of the method

We are interested in two questions: firstly whether the iterative method converges at all, and if so, with what speed. The theory behind these questions goes far beyond this book. Above we remarked that convergence can often be guaranteed for M-matrices; with regard to the convergence speed a full analysis is usually only possible in model cases. For the matrices from BVPs, as described in section 4.2.3, we state without proof that the smallest eigenvalue of the coefficient matrix is $O(h^2)$. The geometric convergence ratio $|\lambda(I - AK^{-1})|$ derived above can then be shown to be as follows:

- For the Jacobi method, the ratio is $1 - O(h^2)$;
- For the Gauss-Seidel iteration it also is $1 - O(h^2)$, but the method converges twice as fast;
- For the SOR method, the optimal omega can improve the convergence factor to $1 - O(h)$.

5.5.5 Jacobi versus Gauss-Seidel and parallelism

Above, we mostly looked at the Jacobi, Gauss-Seidel, and SOR methods from a mathematical perspective. However, such considerations are largely superseded by matters of parallelization on modern computers.

First we observe that all computations in one iteration of the Jacobi method are completely independent, so they can simply be vectorized or done in parallel. That story is different for Gauss-Seidel (and we ignore SOR from now on, since it only differs from Gauss-Seidel in the damping parameter): since the computation of the x_i points of one iteration are now dependent, this type of iteration is not simple to vectorize or to implement on a parallel computer.

In many cases, both methods are considered superseded by the CG or Generalized Minimum Residual (GMRES) methods (sections 5.5.11 and 5.5.13). The Jacobi method is sometimes used as a preconditioner for these methods. One place where Gauss-Seidel is still popular is as a *multigrid smoother*. In that case, parallelism is often found by using a *red-black ordering* of the variables; section 7.8.2.1.

Further discussion of the issue of parallelism in preconditioners can be found in section 7.7.

5.5.6 Choice of K

The convergence and error analysis above showed that the closer K is to A, the faster the convergence will be. In the initial examples we already saw the diagonal and lower triangular choice for K. We can

describe these formally by letting $A = D_A + L_A + U_A$ be a splitting into diagonal, lower triangular, upper triangular part of A. Here are some methods with their traditional names:

- Richardson iteration: $K = \alpha I$.
- Jacobi method: $K = D_A$ (diagonal part),
- Gauss-Seidel method: $K = D_A + L_A$ (lower triangle, including diagonal)
- The SOR method: $K = \omega^{-1} D_A + L_A$
- Symmetric SOR (SSOR) method: $K = (D_A + L_A) D_A^{-1} (D_A + U_A)$.
- In *iterative refinement* we let $K = LU$ be a true factorization of A. In exact arithmetic, solving a system $LUx = y$ gives you the exact solution, so using $K = LU$ in an iterative method would give convergence after one step. In practice, roundoff error will make the solution be inexact, so people will sometimes iterate a few steps to get higher accuracy.

Exercise 5.49. What is the extra cost of a few steps of iterative refinement over a single system solution, assuming a dense system?

Exercise 5.50. The Jacobi iteration for the linear system $Ax = b$ is defined as

$$x_{i+1} = x_i - K^{-1}(Ax_i - b)$$

where K is the diagonal of A. Show that you can transform the linear system (that is, find a different coefficient matrix and right hand side vector that will still have the same solution) so that you can compute the same x_i vectors but with $K = I$, the identity matrix.
What are the implications of this strategy, in terms of storage and operation counts? Are there special implications if A is a sparse matrix?
Suppose A is symmetric. Give a simple example to show that $K^{-1}A$ does not have to be symmetric. Can you come up with a different transformation of the system so that symmetry of the coefficient matrix is preserved and that has the same advantages as the transformation above? You can assume that the matrix has positive diagonal elements.

Exercise 5.51. Show that the transformation of the previous exercise can also be done for the Gauss-Seidel method. Give several reasons why this is not a good idea.

Remark *Stationary iteration can be considered as a form of* inexact Newton's method*, where each iteration uses the same approximation to the inverse of the derivative. Standard functional analysis results [118] state how far this approximation can deviate from the exact inverse.*

A special case is iterative refinement*, where the Newton method should converge in one step, but in practice takes multiple steps because of roundoff in computer arithmetic. The fact that the Newton method will converge as long as the function (or the residual) is calculated accurately enough, can be exploited by doing the LU solution in lower precision, thus getting higher performance [26].*

There are many different ways of choosing the preconditioner matrix K. Some of them are defined algebraically, such as the incomplete factorization discussed below. Other choices are inspired by the differential equation. For instance, if the operator is

$$\frac{\delta}{\delta x}(a(x,y)\frac{\delta}{\delta x}u(x,y)) + \frac{\delta}{\delta y}(b(x,y)\frac{\delta}{\delta y}u(x,y)) = f(x,y)$$

then the matrix K could be derived from the operator

$$\frac{\delta}{\delta x}(\tilde{a}(x)\frac{\delta}{\delta x}u(x,y)) + \frac{\delta}{\delta y}(\tilde{b}(y)\frac{\delta}{\delta y}u(x,y)) = f(x,y)$$

for some choices of $\tilde{a}, \tilde{b}$. The second set of equations is called a *separable problem*, and there are *fast solvers* for them, meaning that they have $O(N \log N)$ time complexity; see [195].

5.5.6.1 Constructing K as an incomplete LU factorization

We briefly mention one other choice of K, which is inspired by Gaussian elimination. As in Gaussian elimination, we let $K = LU$, but now we use an *Incomplete LU (ILU)* factorization. Remember that a regular LU factorization is expensive because of the fill-in phenomenon. In an incomplete factorization, we limit the fill-in artificially.

If we write Gauss elimination as

```
for k,i,j:
   a[i,j] = a[i,j] - a[i,k] * a[k,j] / a[k,k]
```

we define an incomplete variant by

```
for k,i,j:
  if a[i,j] not zero:
    a[i,j] = a[i,j] - a[i,k] * a[k,j] / a[k,k]
```

- The resulting factorization is no longer exact: $LU \approx A$, so it is called an Incomplete LU (ILU) factorization.
- An ILU factorization takes much less space than a full factorization: the sparsity of $L+U$ is the same as of A.

The algorithm above is called 'ILU(0)', where the zero refers to the fact that absolutely no fill-in is allowed during the incomplete factorization. Other schemes that allow a limited amount of fill-in exist. Much more can be said about this method; we will only remark that for *M-matrices* this scheme typically gives a converging method [148].

Exercise 5.52. How do operation counts of the matrix-vector product and solving a system with an ILU factorization compare?
Recall the discussion of parallelism in the Jacobi versus Gauss-Seidel methods; section 5.5.5. How is that for ILU?

You have seen that a full factorization of sparse matrix can need a higher order storage ($N^{3/2}$ for the factorization versus N for the matrix), but that an incomplete factorization takes $O(N)$, just like the matrix. It may therefore come as a surprise that the error matrix $R = A - LU$ is not dense, but itself sparse.

Exercise 5.53. Let A be the matrix of the *Laplacian*, LU an incomplete factorization, and $R = A - LU$. Show that R is a bi-diagonal matrix:

- Consider that R consists of those elements that are discarded during the factorization. Where are they located in the matrix?
- Alternatively, write out the sparsity pattern of the product LU and compare that to the sparsity pattern of A.

5.5.6.2 *Cost of constructing a preconditioner*

In the example of the heat equation (section 4.3) you saw that each time step involves solving a linear system. As an important practical consequence, any setup cost for solving the linear system, such as constructing the preconditioner, will be amortized over the sequence of systems that is to be solved. A similar argument holds in the context of nonlinear equations, a topic that we will not discuss as such. Nonlinear equations are solved by an iterative process such as *Newton's method*, which in its multidimensional form leads to a sequence of linear systems. Although these have different coefficient matrices, it is again possible to amortize setup costs by reusing the preconditioner for a number of Newton steps.

5.5.6.3 *Parallel preconditioners*

Constructing and using a preconditioner is a balancing act of many considerations: a more accurate preconditioner leads to convergence in fewer iterations, but these iterations can be more expensive; in addition, a more accurate preconditioner can be more costly to construct. In parallel, this story is even more complicated, because certain preconditioners are not very parallel to begin with. Therefore, we may accept a preconditioner that is parallel, but that has a worse iteration count than a serial preconditioner. For more discussion, see section 7.7.

5.5.7 Stopping tests

The next question we need to tackle is when to stop iterating. Above we saw that the error decreases geometrically, so clearly we will never reach the solution exactly, even if that were possible in computer arithmetic. Since we only have this relative convergence behavior, how do we know when we are close enough?

We would like the error $e_i = x - x_i$ to be small, but measuring this is impossible. Above we observed that $Ae_i = r_i$, so

$$\|e_i\| \leq \|A^{-1}\|\|r_i\| \leq \lambda_{\max}(A^{-1})\|r_i\|$$

If we know anything about the eigenvalues of A, this gives us a bound on the error. (The norm of A is only the largest eigenvalue for symmetric A. In general, we need singular values here.)

Another possibility is to monitor changes in the computed solution. If these are small:

$$\|x_{n+1} - x_n\|/\|x_n\| < \epsilon$$

we can also conclude that we are close to the solution.

Exercise 5.54. Prove an analytic relationship between the distance between iterates and the distance to the true solution. If your equation contains constants, can they be determined theoretically or in practice?

Exercise 5.55. Write a simple program to experiment with linear system solving. Take the matrix from the 1D BVP (use an efficient storage scheme) and program an iterative method using the choice $K = D_A$. Experiment with stopping tests on the residual and the distance between iterates. How does the number of iterations depend on the size of the matrix?

Change the matrix construction so that a certain quantity is added the diagonal, that is, add αI to the original matrix. What happens when $\alpha > 0$? What happens when $\alpha < 0$? Can you find the value where the behavior changes? Does that value depend on the matrix size?

5.5.8 Theory of general polynomial iterative methods

Above, you saw iterative methods of the form $x_{i+1} = x_i - K^{-1}r_i$, and we will now see iterative methods of the more general form

$$x_{i+1} = x_i + \sum_{j \leq i} K^{-1}r_j\alpha_{ji}, \tag{5.17}$$

that is, using all previous residuals to update the iterate. One might ask, 'why not introduce an extra parameter and write $x_{i+1} = \alpha_i x_i + \cdots$?' Here we give a short argument that the former scheme describes a large class of methods. Indeed, the current author is not aware of methods that fall outside this scheme.

We defined the residual, given an approximate solution $\tilde{x}$, as $\tilde{r} = A\tilde{x} - b$. For this general discussion we precondition the system as $K^{-1}Ax = K^{-1}b$. (See section 5.5.6 where we discussed transforming the linear system.) The corresponding residual for the initial guess $\tilde{x}$ is

$$\tilde{r} = K^{-1}A\tilde{x} - K^{-1}b.$$

We now find that

$$x = A^{-1}b = \tilde{x} - A^{-1}K\tilde{r} = \tilde{x} - (K^{-1}A)^{-1}\tilde{r}.$$

Now, the *Cayley-Hamilton theorem* states that for every A there exists a polynomial $\phi(x)$ (the *characteristic polynomial*) such that

$$\phi(A) = 0.$$

We observe that we can write this polynomial ϕ as

$$\phi(x) = 1 + x\pi(x)$$

where π is another polynomial. Applying this to $K^{-1}A$, we have

$$0 = \phi(K^{-1}A) = I + K^{-1}A\pi(K^{-1}A) \Rightarrow (K^{-1}A)^{-1} = -\pi(K^{-1}A)$$

so that $x = \tilde{x} + \pi(K^{-1}A)\tilde{r}$. Now, if we let $x_0 = \tilde{x}$, then $\tilde{r} = K^{-1}r_0$, giving the equation

$$x = x_0 + \pi(K^{-1}A)K^{-1}r_0.$$

This equation suggests an iterative scheme: if we can find a series of polynomials $\pi^{(i)}$ of degree i to approximate π, it will give us a sequence of iterates

$$x_{i+1} = x_0 + \pi^{(i)}(K^{-1}A)K^{-1}r_0 = x_0 + K^{-1}\pi^{(i)}(AK^{-1})r_0 \tag{5.18}$$

that ultimately reaches the true solution. Based on this use of polynomials in the iterative process, such methods are called *polynomial iterative methods*.

Exercise 5.56. Are stationary iterative methods polynomial methods? Can you make a connection with *Horner's rule*?

Multiplying equation (5.18) by A and subtracting b on both sides gives

$$r_{i+1} = r_0 + \tilde{\pi}^{(i)}(AK^{-1})r_0$$

where $\tilde{\pi}^{(i)}(x) = x\pi^{(i)}(x)$. This immediately gives us

$$r_i = \hat{\pi}^{(i)}(AK^{-1})r_0 \tag{5.19}$$

where $\hat{\pi}^{(i)}$ is a polynomial of degree i with $\hat{\pi}^{(i)}(0) = 1$. This statement can be used as the basis of a convergence theory of iterative methods. However, this goes beyond the scope of this book.

Let us look at a couple of instances of equation (5.19). For $i = 1$ we have

$$r_1 = (\alpha_1 AK^{-1} + \alpha_2 I)r_0 \Rightarrow AK^{-1}r_0 = \beta_1 r_1 + \beta_0 r_0$$

for some values α_i, β_i. For $i = 2$

$$r_2 = (\alpha_2(AK^{-1})^2 + \alpha_1 AK^{-1} + \alpha_0)r_0$$

for different values α_i. But we had already established that AK_0^{-1} is a combination of r_1, r_0, so now we have that

$$(AK^{-1})^2 r_0 \in [\![r_2, r_1, r_0]\!],$$

and it is clear how to show inductively that

$$(AK^{-1})^i r_0 \in [\![r_i, \ldots, r_0]\!]. \tag{5.20}$$

Substituting this in (5.18) we finally get

$$x_{i+1} = x_0 + \sum_{j \le i} K^{-1} r_j \alpha_{ji}. \tag{5.21}$$

It is easy to see that the scheme (5.17) is of the form (5.21) and that the reverse implication also holds.

Summarizing, the basis of iterative methods is a scheme where iterates get updated by all residuals computed so far:

$$x_{i+1} = x_i + \sum_{j \le i} K^{-1} r_j \alpha_{ji}. \tag{5.22}$$

Compare that to the stationary iteration (section 5.5.1) where the iterates get updated from just the last residual, and with a coefficient that stays constant.

We can say more about the α_{ij} coefficients. If we multiply equation (5.22) by A, and subtract b from both sides, we find

$$r_{i+1} = r_i + \sum_{j \le i} AK^{-1} r_j \alpha_{ji}. \tag{5.23}$$

Let us consider this equation for a moment. If we have a starting residual r_0, the next residual is computed as

$$r_1 = r_0 + AK^{-1}r_0\alpha_{00}.$$

From this we get that $AK^{-1}r_0 = \alpha_{00}^{-1}(r_1 - r_0)$, so for the next residual,

$$\begin{aligned} r_2 &= r_1 + AK^{-1}r_1\alpha_{11} + AK^{-1}r_0\alpha_{01} \\ &= r_1 + AK^{-1}r_1\alpha_{11} + \alpha_{00}^{-1}\alpha_{01}(r_1 - r_0) \\ \Rightarrow AK^{-1}r_1 &= \alpha_{11}^{-1}(r_2 - (1 + \alpha_{00}^{-1}\alpha_{01})r_1 + \alpha_{00}^{-1}\alpha_{01}r_0) \end{aligned}$$

We see that we can express $AK^{-1}r_1$ as a sum $r_2\beta_2 + r_1\beta_1 + r_0\beta_0$, and that $\sum_i \beta_i = 0$.

Generalizing this, we find (with different α_{ij} than above)

$$\begin{aligned} r_{i+1} &= r_i + AK^{-1}r_i\delta_i + \textstyle\sum_{j\leq i+1} r_j\alpha_{ji} \\ r_{i+1}(1-\alpha_{i+1,i}) &= AK^{-1}r_i\delta_i + r_i(1+\alpha_{ii}) + \textstyle\sum_{j<i} r_j\alpha_{ji} \\ r_{i+1}\alpha_{i+1,i} &= AK^{-1}r_i\delta_i + \textstyle\sum_{j\leq i} r_j\alpha_{ji} && \text{substituting } \begin{array}{l}\alpha_{ii} := 1+\alpha_{ii} \\ \alpha_{i+1,i} := 1-\alpha_{i+1,i}\end{array} \\ & && \text{note that } \alpha_{i+1,i} = \textstyle\sum_{j\leq i}\alpha_{ji} \\ r_{i+1}\alpha_{i+1,i}\delta_i^{-1} &= AK^{-1}r_i + \textstyle\sum_{j\leq i} r_j\alpha_{ji}\delta_i^{-1} \\ r_{i+1}\alpha_{i+1,i}\delta_i^{-1} &= AK^{-1}r_i + \textstyle\sum_{j\leq i} r_j\alpha_{ji}\delta_i^{-1} \\ r_{i+1}\gamma_{i+1,i} & AK^{-1}r_i + \textstyle\sum_{j\leq i} r_j\gamma_{ji} && \text{substituting } \gamma_{ij} = \alpha_{ij}\delta_j^{-1} \end{aligned}$$

and we have that $\gamma_{i+1,i} = \sum_{j\leq i}\gamma_{ji}$.

We can take this last equation and write it as $AK^{-1}R = RH$ where

$$H = \begin{pmatrix} -\gamma_{11} & -\gamma_{12} & \cdots & & \\ \gamma_{21} & -\gamma_{22} & -\gamma_{23} & \cdots & \\ 0 & \gamma_{32} & -\gamma_{33} & -\gamma_{34} & \\ \emptyset & \ddots & \ddots & \ddots & \ddots \end{pmatrix}$$

In this, H is a so-called *Hessenberg matrix*: it is upper triangular plus a single lower subdiagonal. Also we note that the elements of H in each column sum to zero.

Because of the identity $\gamma_{i+1,i} = \sum_{j\leq i}\gamma_{ji}$ we can subtract b from both sides of the equation for r_{i+1} and 'divide out A', giving

$$x_{i+1}\gamma_{i+1,i} = K^{-1}r_i + \sum_{j\leq i} x_j\gamma_{ji}.$$

This gives us the general form for iterative methods:

$$\begin{cases} r_i = Ax_i - b \\ x_{i+1}\gamma_{i+1,i} = K^{-1}r_i + \sum_{j\leq i} x_j\gamma_{ji} & \text{where } \gamma_{i+1,i} = \sum_{j\leq i}\gamma_{ji}. \\ r_{i+1}\gamma_{i+1,i} = AK^{-1}r_i + \sum_{j\leq i} r_j\gamma_{ji} \end{cases} \tag{5.24}$$

This form holds for many iterative methods, including the stationary iterative methods you have seen above. In the next sections you will see how the γ_{ij} coefficients follow from orthogonality conditions on the residuals.

5.5.9 Iterating by orthogonalization

The stationary methods described above (section 5.5.1) have been around in some form or another for a long time: Gauss described some variant in a letter to a student. They were perfected in the thesis of Young [202] in 1950; the final reference is probably the book by Varga [190]. These methods are little used these days, except in the specialized context of *multigrid smoothers*, a topic not discussed in this course.

At almost the same time, the field of methods based on orthogonalization was kicked off by two papers [131, 102], though it took a few decades for them to find wide applicability. (For further history, see [79].)

The basic idea is as follows:

> If you can make all your residuals orthogonal to each other, and the matrix is of dimension n, then after n iterations you have to have converged: it is not possible to have an $n+1$-st residual that is orthogonal to all previous and nonzero. Since a zero residual means that the corresponding iterate is the solution, we conclude that after n iterations we have the true solution in hand.

With the size of matrices that contemporary applications generate this reasoning is no longer relevant: it is not computationally realistic to iterate for n iterations. Moreover, roundoff will probably destroy any accuracy of the solution. However, it was later realized [168] that such methods *are* a realistic option in the case of *symmetric positive definite (SPD)* matrices. The reasoning is then:

> The sequence of residuals spans a series of subspaces of increasing dimension, and by orthogonalizing, the new residuals are projected on these spaces. This means that they will have decreasing sizes.

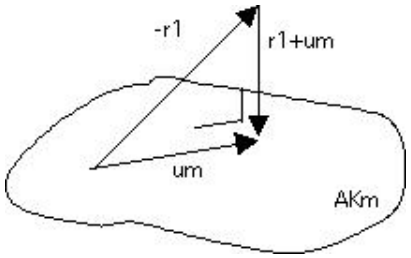


Figure 5.10: The optimal update u_m make the new residual orthogonal to the AK_m subspace.

This is illustrated in figure 5.10.

In this section you will see the basic idea of iterating by orthogonalization. The method presented here is only of theoretical interest; next you will see the Conjugate Gradients (CG) and Generalized Minimum Residual (GMRES) methods that are the basis of many real-life applications.

Let us now take the basic scheme (5.24) and orthogonalize the residuals. Instead of the normal inner product we use the K^{-1}-inner product:

$$(x,y)_{K^{-1}} = x^t K^{-1} y$$

and we will force residuals to be K^{-1}-orthogonal:

$$\forall_{i\neq j}: r_i \perp_{K^{-1}} r_j \Leftrightarrow \forall_{i\neq j}: r_i K^{-1} r_j = 0$$

This is known as the *Full Orthogonalization Method (FOM)* scheme:

Let r_0 be given
For $i \geq 0$:
  let $s \leftarrow K^{-1}r_i$
  let $t \leftarrow AK^{-1}r_i$
  for $j \leq i$:
    let γ_j be the coefficient so that $t - \gamma_j r_j \perp r_j$
  for $j \leq i$:
    form $s \leftarrow s - \gamma_j x_j$
    and $t \leftarrow t - \gamma_j r_j$
  let $x_{i+1} = (\sum_j \gamma_j)^{-1} s$, $r_{i+1} = (\sum_j \gamma_j)^{-1} t$.

You may recognize the *Gram-Schmidt* orthogonalization in this (see appendix 14.2 for an explanation): in each iteration r_{i+1} is initially set to $AK^{-1}r_i$, and orthogonalized against r_j with $j \leq i$.

We can use *modified Gram-Schmidt* by rewriting the algorithm as:

Let r_0 be given
For $i \geq 0$:
  let $s \leftarrow K^{-1}r_i$
  let $t \leftarrow AK^{-1}r_i$
  for $j \leq i$:
    let γ_j be the coefficient so that $t - \gamma_j r_j \perp r_j$
    form $s \leftarrow s - \gamma_j x_j$
    and $t \leftarrow t - \gamma_j r_j$
  let $x_{i+1} = (\sum_j \gamma_j)^{-1} s$, $r_{i+1} = (\sum_j \gamma_j)^{-1} t$.

These two version of the FOM algorithm are equivalent in exact arithmetic, but differ in practical circumstances in two ways:

- The modified Gram-Schmidt method is more numerically stable;
- The unmodified method allows you to compute all inner products simultaneously, which cuts down on network latency. We discuss this below in sections 7.1.3, 7.6.1.2, 7.6.

Even though the FOM algorithm is not used in practice, these computational considerations carry over to the GMRES method below.

5.5.10 Coupled recurrences form of iterative methods

Above, you saw the general equation (5.24) for generating iterates and search directions. This equation is often split as

- An update of the x_i iterate from a single *search direction*:

$$x_{i+1} = x_i - \delta_i p_i,$$

and
- A construction of the search direction from the residuals known so far:

$$p_i = K^{-1} r_i + \sum_{j<i} \beta_{ij} K^{-1} r_j.$$

It is not hard to see inductively that we can also define

$$p_i = K^{-1} r_i + \sum_{j<i} \gamma_{ji} p_j,$$

and this last form is the one that is used in practice.

The iteration dependent coefficients are typically chosen to let the residuals satisfy various orthogonality conditions. For instance, one can choose to let the method be defined by letting the residuals be orthogonal ($r_i^t r_j = 0$ if $i \neq j$), or A-orthogonal ($r_i^t A r_j = 0$ if $i \neq j$). Many more schemes exist. Such methods can converge much faster than stationary iteration, or converge for a wider range of matrix and preconditioner types. Below we will see two such methods; their analysis, however, is beyond the scope of this course.

5.5.11 The method of Conjugate Gradients

In this section, we will derive the Conjugate Gradients (CG) method, which is a specific implementation of the FOM algorithm. In particular, it has pleasant computational properties in the case of an SPD matrix A.

The CG method takes as its basic form the coupled recurrences formulation described above, and the coefficients are defined by demanding that the sequence of residuals $r_0, r_1, r_2, \ldots$ satisfy

$$r_i^t K^{-1} r_j = 0 \quad \text{if } i \neq j.$$

We start by deriving the CG method for nonsymmetric systems, and then show how it simplifies in the symmetric case. (The approach here is taken from [58]).

The basic equations are

$$\begin{cases} x_{i+1} = x_i - \delta_i p_i \\ r_{i+1} = r_i - \delta_i A p_i \\ p_{i+1} = K^{-1} r_{i+1} + \sum_{j \leq i} \gamma_{ji+1} p_j, \end{cases} \tag{5.25}$$

where the first and third equation were introduced above, and the second can be found by multiplying the first by A (check this!).

We will now derive the coefficients in this method by induction. In essence, we assume that we have current residual r_{cur}, a residuals to be computed r_{new}, and a collection of known residuals R_{old}. Rather than using subscripts 'old, cur, new', we use the following convention:

- x_1, r_1, p_1 are the current iterate, residual, and search direction. Note that the subscript 1 does not denote the iteration number here.
- x_2, r_2, p_2 are the iterate, residual, and search direction that we are about to compute. Again, the subscript does not equal the iteration number.
- X_0, R_0, P_0 are all previous iterates, residuals, and search directions bundled together in a block of vectors.

In terms of these quantities, the update equations are then

$$\begin{cases} x_2 = x_1 - \delta_1 p_1 \\ r_2 = r_1 - \delta_i A p_1 \\ p_2 = K^{-1} r_2 + \upsilon_{12} p_1 + P_0 u_{02} \end{cases} \tag{5.26}$$

where δ_1, υ_{12} are scalars, and u_{02} is a vector with length the number of iterations before the current. We now derive $\delta_1, \upsilon_{12}, u_{02}$ from the orthogonality of the residuals. To be specific, the residuals have to be orthogonal under the K^{-1} inner product: we want to have

$$r_2^t K^{-1} r_1 = 0, \qquad r_2^t K^{-1} R_0 = 0.$$

Combining these relations gives us, for instance,

$$\left.\begin{array}{l} r_1^t K^{-1} r_2 = 0 \\ r_2 = r_1 - \delta_i A K^{-1} p_1 \end{array}\right\} \Rightarrow \delta_1 = \frac{r_1^t K^{-1} r_1}{r_1^t K^{-1} A p_1}.$$

Finding υ_{12}, u_{02} is a little harder. For this, we start by summarizing the relations for the residuals and search directions in equation (5.25) in block form as

$$(R_0, r_1, r_2) \left(\begin{array}{ccccc} 1 & & & & \\ -1 & 1 & & & \\ & \ddots & \ddots & & \\ \hline & & -1 & 1 & \\ \hline & & & -1 & 1 \end{array}\right) = A(P_0, p_1, p_2) \operatorname{diag}(D_0, d_1, d_2)$$

$$(P_0, p_1, p_2) \begin{pmatrix} I - U_{00} & -u_{01} & -u_{02} \\ & 1 & -\upsilon_{12} \\ & & 1 \end{pmatrix} = K^{-1}(R_0, r_1, r_2)$$

or abbreviated $RJ = APD$, $P(I-U) = R$ where J is the matrix with identity diagonal and minus identity subdiagonal. We then observe that

- $R^t K^{-1} R$ is diagonal, expressing the orthogonality of the residuals.
- Combining that $R^t K^{-1} R$ is diagonal and $P(I-U) = R$ gives that $R^t P = R^t K^{-1} R (I-U)^{-1}$. We now reason that $(I-U)^{-1}$ is upper diagonal, so $R^t P$ is upper triangular. This tells us quantities such as $r_2^t p_1$ are zero.
- Combining the relations for R and P, we get first that

$$R^t K^{-t} A P = R^t K^{-t} R J D^{-1}$$

which tells us that $R^t K^{-t} AP$ is lower bidiagonal. Expanding R in this equation gives

$$P^t AP = (I-U)^{-t} R^t RJD^{-1}.$$

Here D and $R^t K^{-1} R$ are diagonal, and $(I-U)^{-t}$ and J are lower triangular, so $P^t AP$ is lower triangular.

- This tells us that $P_0^t A p_2 = 0$ and $p_1^t A p_2 = 0$.
- Taking the product of $P_0^t A$, $p_1^t A$ with the definition of p_2 in equation (5.26) gives

$$u_{02} = -(P_0^t AP_0)^{-1} P_0^t AK^{-1} r_2, \qquad v_{12} = -(p_1^t Ap_1)^{-1} p_1^t AK^{-1} r_2.$$

- If A is symmetric, $P^t AP$ is lower triangular (see above) and symmetric, so it is in fact diagonal. Also, $R^t K^{-t} AP$ is lower bidiagonal, so, using $A = A^t$, $P^t AK^{-1} R$ is upper bidiagonal. Since $P^t AK^{-1} R = P^t AP(I-U)$, we conclude that $I-U$ is upper bidiagonal, so, only in the symmetric case, $u_{02} = 0$.

Some observations about this derivation.

- Strictly speaking we are only proving necessary relations here. It can be shown that these are sufficient too.
- There are different formulas that wind up computing the same vectors, in exact arithmetic. For instance, it is easy to derive that $p_1^t r_1 = r_1^t r_1$, so this can be substituted in the formulas just derived. The implementation of the CG method as it is typically implemented, is given in figure 5.11.
- In the k-th iteration, computing $P_0^t Ar_2$ (which is needed for u_{02}) takes k inner products. First of all, inner products are disadvantageous in a parallel context. Secondly, this requires us to store all search directions indefinitely. This second point implies that both work and storage go up with the number of iterations. Contrast this with the stationary iteration scheme, where storage was limited to the matrix and a few vectors, and work in each iteration was the same.
- The objections just raised disappear in the symmetric case. Since u_{02} is zero, the dependence on P_0 disappears, and only the dependence on p_1 remains. Thus, storage is constant, and the amount of work per iteration is constant. The number of inner products per iteration can be shown to be just two.

Exercise 5.57. Do a flop count of the various operations in one iteration of the CG method. Assume that A is the *matrix of a five-point stencil* and that the preconditioner M is an incomplete factorization of A (section 5.5.6.1). Let N be the matrix size.

5.5.12 Derivation from minimization

The above derivation of the CG method is not often found in the literature. The typical derivation starts with a minimization problem with a *symmetric positive definite (SPD)* matrix A:

$$\text{For which vector } x \text{ with } \|x\| = 1 \text{ is } f(x) = 1/2\, x^t Ax - b^t x \text{ minimal?} \tag{5.27}$$

If we accept the fact that the function f has a minimum, which follows from the positive definiteness, we find the minimum by computing the derivative

$$f'(x) = Ax - b.$$

and asking where $f'(x) = 0$. And, presto, there we have the original linear system.

Compute $r^{(0)} = Ax^{(0)} - b$ for some initial guess $x^{(0)}$
for $i = 1, 2, \ldots$
  solve $Kz^{(i-1)} = r^{(i-1)}$
  $\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$
  if $i = 1$
    $p^{(1)} = z^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\delta_i = \rho_{i-1}/p^{(i)^T} q^{(i)}$
  $x^{(i)} = x^{(i-1)} - \delta_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \delta_i q^{(i)}$
  check convergence; continue if necessary
end

Figure 5.11: The Preconditioned Conjugate Gradient Method.

Exercise 5.58. Derive the derivative formula above. (Hint: write out the definition of derivative as $\lim_{h \downarrow 0} \ldots$.) Note that this requires A to be symmetric.

For the derivation of the iterative method, we state that the iterate x_i is updated with a certain step size δ_i along a *search direction* p_i:

$$x_{i+1} = x_i + p_i \delta_i$$

The optimal *step size*

$$\delta_i = \frac{r_i^t p_i}{p_1^t A p_i}$$

is then derived as the one that minimizes the function f along the line $x_i + \delta\delta p_i$:

$$\delta_i = \operatorname*{argmin}_{\delta} \|f(x_i + p_i \delta)\|$$

The construction of the search direction from the residuals follows by induction proof from the requirement that the residuals be orthogonal. For a typical proof, see [4].

5.5.13 GMRES

In the discussion of the CG method above, it was pointed out that orthogonality of the residuals requires storage of all residuals, and k inner products in the k'th iteration. Unfortunately, it can be proved that the work savings of the CG method can, for all practical purposes, not be found outside of *SPD* matrices [60].

The *GMRES* method is a popular implementation of such full orthogonalization schemes. In order to keep the computational costs within bounds, it is usually implemented as a *restarted* method. That is, only a certain number (say $k = 5$ or 20) of residuals is retained, and every k iterations the method is restarted.

Other methods exist that do not have the growing storage demands of GMRES, for instance QMR [67] and BiCGstab [188]. Even though by the remark above these can not orthogonalize the residuals, they are still attractive in practice.

5.5.14 Convergence and complexity

The efficiency of Gaussian elimination was fairly easy to assess: factoring and solving a system takes, deterministically, $\frac{1}{3}n^3$ operations. For an iterative method, the operation count is the product of the number of operations per iteration times the number of iterations. While each individual iteration is easy to analyze, there is no good theory to predict the number of iterations. (In fact, an iterative method may not even converge to begin with.)

The one basic theorem is that for CG

$$\#it \sim \sqrt{\kappa(A)}$$

while for second-order PDEs

$$\kappa(A) \sim h^{-2} = N,$$

with N the matrix size.

The influence of the preconditioner is harder to quantify. For many preconditioners, such as ILU,

$$\kappa(M^{-1}A) \sim N,$$

but with a lower proportionality constant. Much research has gone into preconditioners that achieve an order improvement:

$$\kappa(M^{-1}A) \sim \sqrt{N}.$$

However, this is often only provable in restricted cases, such as M-matrices [91, 10]. Even more restricted is *multigrid*[22], which ideally can achieve

$$\kappa(M^{-1}A) = O(1).$$

A final consideration in evaluation the efficiency of iterative methods related to processor utilization. Gaussian elimination can be coded in such a way that there is considerable cache reuse, making the algorithm run at a fair percentage of the computer's peak speed. Iterative methods, on the other hand, are much slower on a flops per second basis.

All these considerations make the application of iterative methods to linear system solving somewhere in between a craft and a black art. In practice, people do considerable experimentation to decide whether an iterative method will pay off, and if so, which method is preferable.

5.6 Eigenvalue methods

In this chapter we have so far limited ourselves to linear system solving. *Eigenvalue problems* are another important category of linear algebra applications, but their interest lies more in the mathematics than the computations as such. We give a brief outline of the type of computations involved.

5.6.1 Power method

The *power method* is a simple iteration process (see appendix 14.3 for details): given a matrix A and an arbitrary starting vector v, compute repeatedly

$$v \leftarrow Av, \quad v \leftarrow v/\|v\|.$$

The vector v quickly becomes the eigenvector corresponding to the eigenvalue with maximum absolute size, and so $\|Av\|/\|v\|$ becomes an approximation to that largest eigenvalue.

Applying the power method to A^{-1} is known as *inverse iteration* and it yields the inverse of the eigenvalue that is smallest in absolute magnitude.

Another variant of the power method is the *shift-and-inverse iteration* which can be used to find interior eigenvalues. If σ is close to an interior eigenvalue, then inverse iteration on $A - \sigma I$ will find that interior eigenvalue.

5.6.2 Orthogonal iteration schemes

The fact that eigenvectors to different eigenvalues are orthogonal can be exploited. For instance, after finding one eigenvector, one could iterate in the subspace orthogonal to that. Another option is to iterate on a block of vectors, and orthogonalizing this block after each power method iteration. This produces as many dominant eigenvalue as the block size. The *restarted Arnoldi* method [136] is an example of such a scheme.

5.6.3 Full spectrum methods

The iterative schemes just discussed yield only localized eigenvalues. Other methods compute the full spectrum of a matrix. The most popular of these is the *QR method*.

5.7 Further Reading

Iterative methods is a very deep field. As a practical introduction to the issues involved, you can read the 'Templates book' [8], online at `http://netlib.org/templates/`. For a deeper treatment of the theory, see the book by Saad [169] of which the first edition can be downloaded at `http://www-users.cs.umn.edu/~saad/books.html`.

Chapter 6

Programming for performance

In this section we will look at how different ways of programming can influence the performance of a code. This will only be an introduction to the topic.

The full listings of the codes and explanations of the data graphed here can be found in chapter 29.

6.1 Peak performance

For marketing purposes, it may be desirable to define a 'top speed' for a processor. Since a pipelined floating point unit can yield one result per cycle asymptotically, you would calculate the theoretical *peak performance* as the product of the clock speed (in ticks per second), the width of vector instructions, the number of floating point units, and the number of cores. This top speed is unobtainable in practice, and very few codes come even close to it.

One of the few operations that can come close to peak performance is the *matrix-matrix product*, if properly coded; see section 7.4.1. This is the basis of the *Linpack benchmark*; the parallel version of this benchmark is reported in the 'top 500'; see section 2.11.4.

In a way, the rest of this chapter is an exploration of all the factors that stand in the way of obtaining peak performance.

6.2 Bandwidth

It has been a basic fact of processors of at least the past three decades that the processing unit is faster than the memory: the processor can operate on more numbers per second than the memory can deliver to the processor.

This means that on cache-contained datasets performance is limited by the bandwidth from that cache, and for streaming operations such as operations on long vectors by the bandwidth from main memory. If we would plot performance as a function of dataset size, you would expect a step graph, with larger datasets at a lower bandwidth and performance level.

Let's make an attempt to show such behavior. You'd think you could measure bandwidth by executing a simple *streaming kernel*:

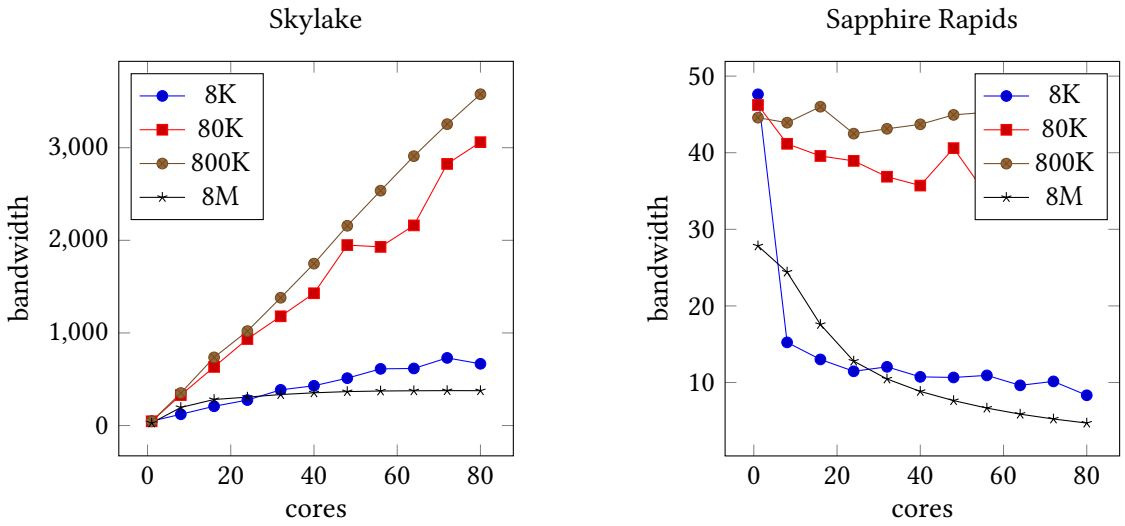


Figure 6.1: Intel Icelake bandwidth (Gbyte/sec) as a function of core counts and data set sizes, left: Aggregate, right: Per-core

```
// hardware/allocation.cpp
template< typename R >
R sum_stream( span<R> stream ) {
  R sum{static_cast<R>(0)};
  for ( size_t i=0; i<stream.size(); ++i )
    sum += stream[i];
  return sum;
};
```

This loads a linear stretch of memory once, and generates only one number in return, We easily compute the bandwidth as in the number of bytes loaded divided by the execution time.

However, there are several factors that complicate this story.

1. The length of the stream is relevant: if the stream is short, it may fit in a cache and a higher bandwidth will be measured then if it has to come from main memory.
2. Modern processors have multiple cores, and bandwidth to main memory is shared between them. Running the streaming kernel on only one core is likely to give a higher per-core bandwidth than when all cores are running the kernel simultaneously.

To get a realistic evaluation, we instead look at the aggregate bandwidth over all available cores.

6.2.1 Aggregate bandwidth

We explore the above factors by running the streaming kernel on multiple stream lengths and multiple core counts.

```
// hardware/bandwidth.cpp
```

```
vector<real> results(nthreads,0.);
float bw{0};
uint64_t data_stream_size{0};
#pragma omp parallel proc_bind(spread) reduction(+:bw)
{
  int my_thread_number = omp_get_thread_num();
  std::span<real> thread_data_stream = storage.get_stream(my_thread_number);
  for (int irepeat=0; irepeat<how_many_repeats; ++irepeat) {
    results.at(my_thread_number) += sum_stream(thread_data_stream);
  }
  bw += how_many_repeats * thread_data_stream.size()*sizeof(real);
}
```

We test this on the *TACC Stampede3* cluster, with dual-socket *Intel Ice Lake* processors, with a total of 112 cores per node. Figure 6.1 shows that for any dataset size the aggregate bandwidth grows with the core count, but the individual per-core bandwidth goes down. With the largest dataset size, aggregate bandwidth does not vary much, because the data streams from main memory. The next two small sizes probably fit in cache, so they show a higher aggregate bandwidth than from main memory. The smallest dataset size is anomalous: for all but the single-core case it is probably the thread synchronization that overwhelms the actual performance.

6.2.2 Striding

In more traditional programming terms:

```
for ( size_t i=0; i<cachesize_in_words; i+=stride )
   f( thecache[i] );
```

Since data is moved from memory to cache in consecutive chunks named cachelines (see section 1.3.5.8), code that does not utilize all data in a cacheline pays a bandwidth penalty. To explore this, we apply a strided operation

```
// hardware/allocation.cpp
template <typename R>
void Cache<R>::transform_in_place
    ( std::function< void(R&) > f,int stride,int nrepeats ) {
  for ( int irepeat=0; irepeat<nrepeats; irepeat++ ) {
    for ( size_t i=0; i<stream_cache_in_words; i+=stride )
      f( thecache[i] );
  }
};
```

for multiple stride values.

As above, we try multiple stream sizes and core counts:

```
// hardware/stride.cpp
#pragma omp parallel proc_bind(spread)
{
  int my_thread_number = omp_get_thread_num();
  auto my_data_stream = write_data.get_stream(my_thread_number);
  auto my_flush_stream = flush_data.get_stream(my_thread_number);
  for (int irepeat=0; irepeat<how_many_repeats; ++irepeat) {
```

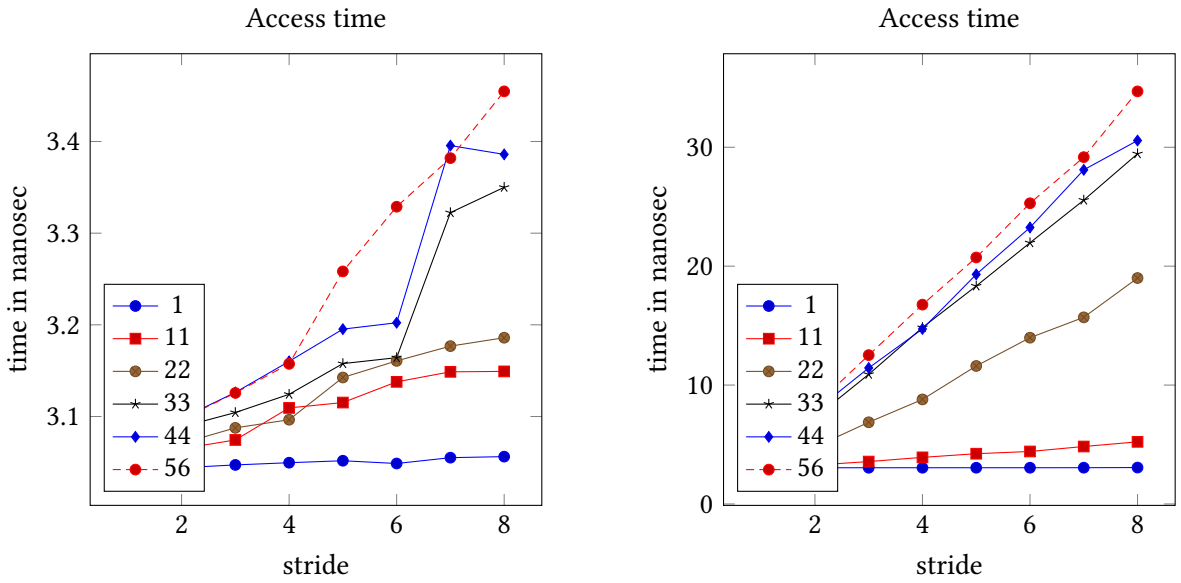


Figure 6.2: Access time per word for different strides and core counts, Frontera; dataset size left: 100k, right 1M; note different y scales!

```
    my_data_stream.transform_in_place( [] (floattype& x ) { x+=1.; }, stride );
  }
}
```

Here we use a constant data set size s, and we compute the effective bandwidth from the number of elements processed $d = s/stride$. Figure 6.2 shows that for a small dataset the access time per element is roughly constant because the data is streamed at high speed from cache. Only at large strides does the bandwdith demand increase sufficiently to become a factor.

However, for a larger data set data is streamed from main memory which does not have enough bandwidth for all cores. We then see the access time per word go up with the stride size.

For more, see section 1.6.1.4.

6.3 Arithmetic performance

6.3.1 Computing with underflow

Many processors will happily compute with denormals, but they have to emulate these computations in *micro-code*. This severely depresses performance.

As an example, we compute a geometric sequence $n \mapsto r^n$ with $r < 1$. For small enough values r, this sequence underflows, and the computation becomes slow. To get macroscopic timings, in this code we actually operate on an array of identical numbers.

	Intel Cascade Lake	AMD Milan	MBP M2
Flush-to-zero	1.5	1.4	1.2
No Ftz	45	1.7	47

Figure 6.3: Access time per word for subormal computing in different processors

```
// hardware/denormal.cpp
storage.set(startvalue);
/* repeated scale the array */
for (int r=0; r<repeats; ++r) {
  storage.transform_in_place( [ratio] (floattype &x) { x *= ratio; } );
  storage.front() = storage.back();
}
```

where the transform routine is:

```
// hardware/allocation.cpp
template <typename R>
void Cache<R>::transform_in_place
    ( std::function< void(R&) > f,int stride,int nrepeats ) {
  for ( int irepeat=0; irepeat<nrepeats; irepeat++ ) {
    for ( size_t i=0; i<stream_cache_in_words; i+=stride )
      f( thecache[i] );
  }
};
```

First we use the default behavior of *flush-to-zero* where any subnormal number is set to zero; this is specified by a compiler flag `-ftz`. Then we show the slowdown associated with correct handling of denormals, specified by a compiler flag `-no-ftz`. Table 6.3 reports both modes in an *Intel Cascade Lake*, an *AMD Milan*, and an *Apple M2* processor. We see that 'true' handling of denormals can have severe performance penalties.

6.3.2 Pipelining

In section 1.2.1.3 you learned that the floating point units in a modern CPU are pipelined, and that pipelines require a number of independent operations to function efficiently. The typical pipelineable operation is a vector addition; an example of an operation that can not be pipelined is the inner product accumulation

```
for (i=0; i<N; i++)
  s += a[i]*b[i];
```

The fact that `s` gets both read and written halts the addition pipeline. One way to fill the *floating point pipeline* is to apply *loop unrolling*:

```
for (i = 0; i < N/2-1; i ++) {
  sum1 += a[2*i] * b[2*i];
  sum2 += a[2*i+1] * b[2*i+1];
}
```

Now there are two independent multiplies in between the accumulations. With a little indexing optimization this becomes:

```
for (i = 0; i < N/2-1; i ++) {
  sum1 += *(a + 0) * *(b + 0);
  sum2 += *(a + 1) * *(b + 1);

  a += 2; b += 2;
}
```

In a further optimization, we disentangle the addition and multiplication part of each instruction. The hope is that while the accumulation is waiting for the result of the multiplication, the intervening instructions will keep the processor busy, in effect increasing the number of operations per second.

```
for (i = 0; i < N/2-1; i ++) {
  temp1 = *(a + 0) * *(b + 0);
  temp2 = *(a + 1) * *(b + 1);

  sum1 += temp1; sum2 += temp2;

  a += 2; b += 2;
}
```

Finally, we realize that the furthest we can move the addition away from the multiplication, is to put it right in front of the multiplication *of the next iteration*:

```
for (i = 0; i < N/2-1; i ++) {
  sum1 += temp1;
  temp1 = *(a + 0) * *(b + 0);

  sum2 += temp2;
  temp2 = *(a + 1) * *(b + 1);

  a += 2; b += 2;
}
s = temp1 + temp2;
```

Of course, we can unroll the operation by more than a factor of two. While we expect an increased performance because of the longer sequence of pipelined operations, large unroll factors need large numbers of registers. Asking for more registers than a CPU has is called *register spill*, and it will decrease performance.

Another thing to keep in mind is that the total number of operations is unlikely to be divisible by the unroll factor. This requires *cleanup code* after the loop to account for the final iterations. Thus, unrolled code is harder to write than straight code, and people have written tools to perform such *source-to-source transformations* automatically.

6.3.3 Semantics of unrolling

An observation about the unrollling transformation is that we are implicitly using associativity and commutativity of addition: while the same quantities are added, they are now in effect added in a different order. As you will see in chapter 3, in computer arithmetic this is not guaranteed to give the exact same result.

For this reason, a compiler will only apply this transformation if explicitly allowed. For instance, for the *Intel compiler*, the option `fp-model precise` indicates that code transformations should preserve the semantics of floating point arithmetic, and unrolling is not allowed. On the other hand `fp-model fast` indicates that floating point arithmetic can be sacrificed for speed.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                Processing 16384 words
                using memory model: fast
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Unroll  1.. ran for      518 usec (efficiency: 100% )
Unroll  2.. ran for      451 usec (efficiency: 114% )
Unroll  4.. ran for      258 usec (efficiency: 200% )
unroll factor 8 not implemented
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                Processing 16384 words
                using memory model: precise
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Unroll  1.. ran for     1782 usec (efficiency: 100% )
Unroll  2.. ran for      444 usec (efficiency: 401% )
Unroll  4.. ran for      256 usec (efficiency: 696% )
unroll factor 8 not implemented
```

6.4 Hardware exploration

6.4.1 Cache size

Above you learned that data from L1 can be processed with lower latency and higher bandwidth than from L2, and L2 is again faster than L3 or memory. Therefore, if a code has the potential to reuse data you want to write it in such a way that reused data fits in cache at some level, the higher – meaning closer to the process – the better.

As a prototypical example, consider code that repeatedly accesses the same data:

```
for (int i=0; i<NRUNS; i++)
  for (int j=0; j<datasize; j++)
    array_data[j] = 2.3*array_data[j]+1.2;
```

If the size parameter allows the array to fit in cache, the operation will be relatively fast. As the size of the dataset grows, parts of it will evict other parts from the L1 cache, so the speed of the operation will be determined by the latency and bandwidth of the L2 cache.

Exercise 6.1. Argue that with a large enough problem and an LRU replacement policy (section 1.3.5.7) essentially all data in the L1 will be replaced in every iteration of the outer loop. Can you write an example code that will let some of the L1 data stay resident?

It may be possible to arrange the operations to keep data in L1 cache. For instance, in our example, we could write

```
blockstart = 0;
for (int b=0; b<datasize/l1size; b++) {
  for (int i=0; i<NRUNS; i++) {
```

```
    for (int j=0; j<l1size; j++)
      array_data[blockstart+j] = 2.3*array_data[blockstart+j]+1.2;
  }
  blockstart += l1size;
}
```

assuming that the L1 size divides evenly in the dataset size. This strategy is called *cache blocking* or *blocking for cache reuse*.

Remark *Like unrolling, blocking code may change the order of evaluation of expressions. Since floating point arithmetic is not* associative*, blocking is not a transformation that compilers are always allowed to make. You can supply compiler options to indicate whether the lanuage rules are to be strictly adhered to.*

6.4.2 Demonstrating the influence of cache on performance

While blocking is an essential technique in operations such as the matrix-matrix product kernel, the above example is too simple. In this section we explore blocking in more detail.

You can try to write a loop over a small array as above, and execute it many times, hoping to observe performance degradation when the array gets larger than the cache size. The number of iterations should be chosen so that the measured time is well over the resolution of your clock.

This runs into a couple of unforeseen problems. The timing for a simple loop nest

```
for (int irepeat=0; irepeat<how_many_repeats; irepeat++) {
  for (int iword=0; iword<cachesize_in_words; iword++)
    data_array[iword] += 1.1;
}
```

may seem to depend linearly on the array size, without any influence from being cache contained.

To see what the compiler does with this fragment, let your compiler generate an *optimization report*. (For the *Intel compiler* use `-qopt-report`.) In this report you see that the compiler has decided to exchange the loops: Each element of the array is then loaded only once.

```
remark #25444: Loopnest Interchanged: ( 1 2 ) --> ( 2 1 )
remark #15542: loop was not vectorized: inner loop was already vectorized
```

Now it is going over the array just once, executing an accumulation loop on each element. Here the cache size is indeed irrelevant.

In an attempt to find an example that does not show this loop exchange and observe cache influence you can try to make the inner loop too complicated for the compiler to analyze, as would be the case in applications for irregular access patterns. You could for instance turn the array into a sort of *linked list* that you traverse:

```
// setup
for (int iword=0; iword<cachesize_in_words; iword++)
    data_array[iword] = (iword+1) % cachesize_in_words

// use:
```

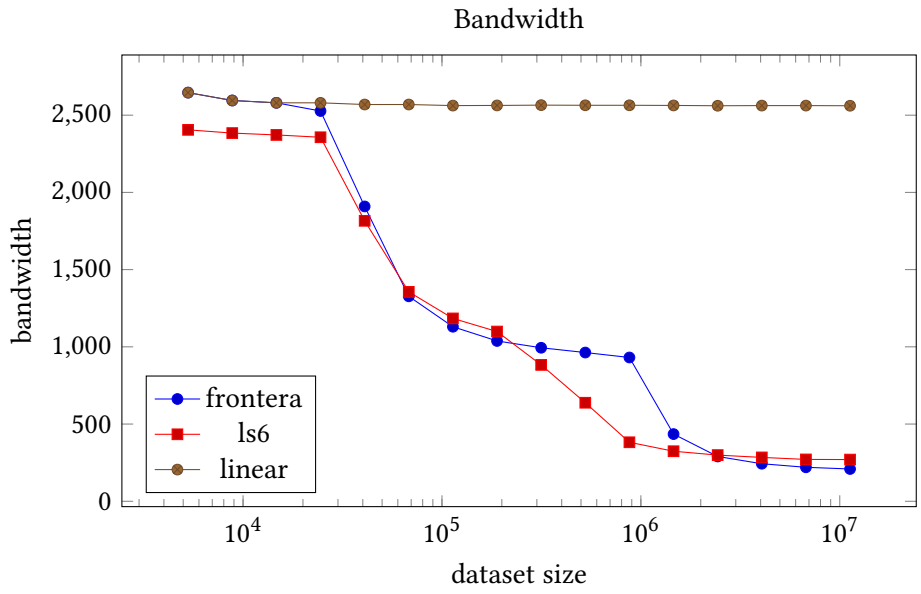


Figure 6.4: Single core bandwidth as function of data set size

```
for ( /* several steps */ ) {
  ptr = 0;
  for (int iword=0; iword<cachesize_in_words; iword++)
      ptr = data_array[ptr];
```

Now the compiler will not exchange the loops, but you will still not observe the cache size threshold. The reason for this is that with regular access the *memory prefetcher* (section 1.3.6) kicks in: some component of the CPU predicts what address(es) you will be requesting next, and fetches it/them in advance.

To stymie this bit of cleverness you need to make the linked list more random:

```
for (int iword=0; iword<cachesize_in_words; iword++)
    data_array[iword] = random() % cachesize_in_words
```

Given a sufficiently large cachesize this will be a cycle that touches all array locations, or you can explicitly ensure this by generating a permutation of all index locations.

We illustrate this in figure 6.4. For the *TACC Frontera* machine with *Intel Cascade Lake* processors we plot the linear traversal (top line) and the random traversal. For the *TACC Lonestar6* machine with *AMD Milan* processors we plot random traversal. The dataset sizes corresponding to L1 and L2 caches are clearly visible. Exceeding the L2 cache has a drop-off in performance, but not as drastic as for the L1 cache.

Let's recap what we have learned here. First of all, cachesize will *a priori* have an influence on code performance: the more cache-contained, the better. But secondly: if the code mostly traverses data linearly, the prefetcher makes the cache effects felt far less; only code that jumps somewhat randomly through its data is sensitive to being cache-contained.

Exercise 6.2. While the strategy just sketched will demonstrate the existence of cache sizes, it will not report the maximal bandwidth that the cache supports. What is the problem and how would you fix it?

6.4.3 Detailed timing

If we have access to a cycle-accurate timer or the hardware counters, we can actually plot the number of cycles per access.

Such a plot is given in figure 6.5. The full code is given in section 29.2.

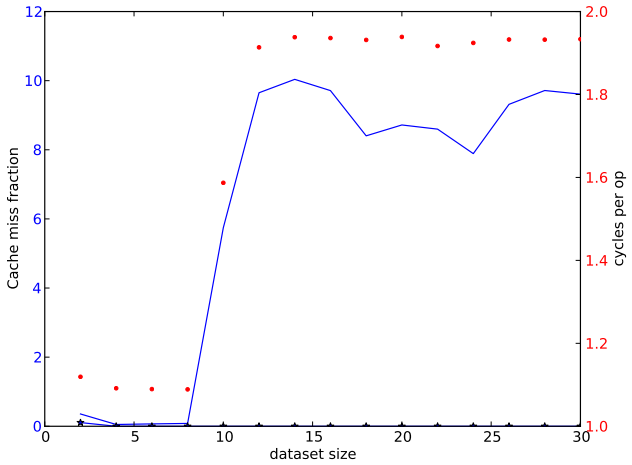


Figure 6.5: Average cycle count per operation as function of the dataset size.

6.4.4 TLB

As explained in section 1.3.9.2, the Translation Look-aside Buffer (TLB) maintains a small list of frequently used memory pages and their locations; addressing data that are location on one of these pages is much faster than data that are not. Consequently, one wants to code in such a way that the number of pages accessed is kept low.

Consider code for traversing the elements of a two-dimensional array in two different ways.

```
#define INDEX(i,j,m,n) i+j*m
array = (double*) malloc(m*n*sizeof(double));

/* traversal #1 */
for (j=0; j<n; j++)
  for (i=0; i<m; i++)
    array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;

/* traversal #2 */
```

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
```

The results (see Appendix 29.5 for the source code) are plotted in figures 6.7 and 6.6.

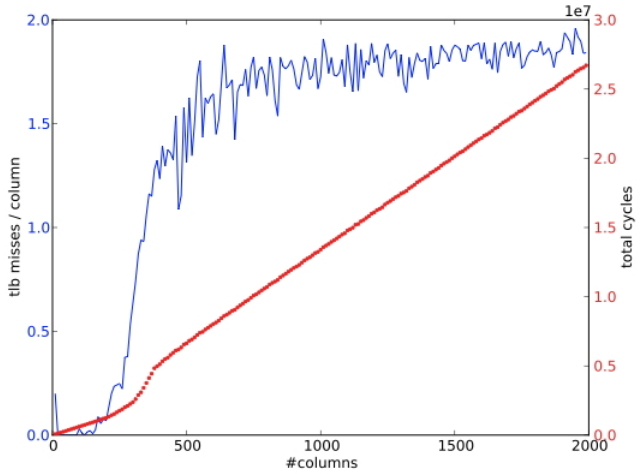


Figure 6.6: Number of TLB misses per column as function of the number of columns; columnwise traversal of the array.

Using $m = 1000$ means that, on the *AMD Opteron* which has pages of 512 doubles, we need roughly two pages for each column. We run this example, plotting the number 'TLB misses', that is, the number of times a page is referenced that is not recorded in the TLB.

1. In the first traversal this is indeed what happens. After we touch an element, and the TLB records the page it is on, all other elements on that page are used subsequently, so no further TLB misses occur. Figure 6.6 shows that, with increasing n, the number of TLB misses per column is roughly two.
2. In the second traversal, we touch a new page for every element of the first row. Elements of the second row will be on these pages, so, as long as the number of columns is less than the number of TLB entries, these pages will still be recorded in the TLB. As the number of columns grows, the number of TLB increases, and ultimately there will be one TLB miss for each element access. Figure 6.7 shows that, with a large enough number of columns, the number of TLB misses per column is equal to the number of elements per column.

6.5 Cache associativity

There are many algorithms that work by recursive division of a problem, for instance the *FFT* algorithm. As a result, code for such algorithms will often operate on vectors whose length is a power of two. Unfortunately, this can cause conflicts with certain architectural features of a CPU, many of which involve powers of two.

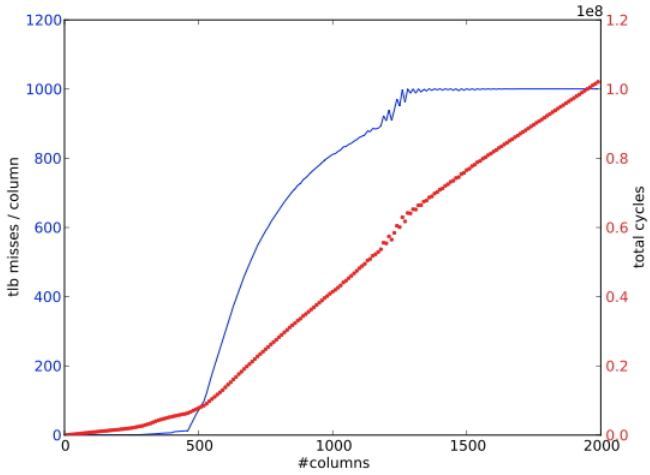


Figure 6.7: Number of TLB misses per column as function of the number of columns; rowwise traversal of the array.

In section 1.3.5.10 you saw how the operation of adding a small number of vectors

$$\forall_j : y_j = y_j + \sum_{i=1}^{m} x_{i,j}$$

is a problem for direct mapped caches or set-associative caches with associativity.

We simplify this into a code that cycles through a number of locations in the same associativity equivalence class:

```
// hardware/associativity.cpp
auto data = write_data.get_stream().data();
for ( int irepeat=0; irepeat<how_many_repeats; ++irepeat )
  for ( int loc=0; loc<data.size(); loc+=displacement_in_words )
    data[loc] += loc;
```

where the data size has been computed to accomodated a cycle of length `assoc`:

```
// hardware/associativity.cpp
auto datasize_in_bytes = displacement_in_bytes * assoc;
auto datasize_in_words = datasize_in_bytes / sizeof(floattype);
```

To evaluate the effects of associativity we measure the access time per element on various generations of Intel processors. Figure 6.8 shows the *Intel Sky Lake* and *Intel Cascade Lake*, both with a 32KiB L1 cache that is 8-way associative, and the *Intel Ice Lake* with a 48KiB L1 cache that is 12-way associative. In all cases we see that the access time per element goes up dramatically when cache associativity is exceeded.

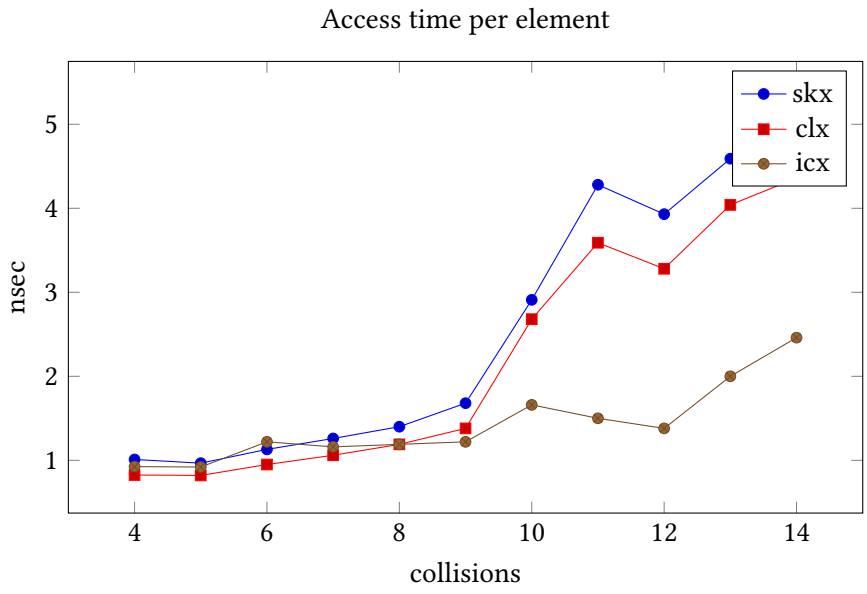


Figure 6.8: Effects of cache associativity

6.6 Loop nests

If your code has *nested loops*, and the iterations of the outer loop are independent, you have a choice which loop to make outer and which to make inner.

Exercise 6.3. Give an example of a doubly-nested loop where the loops can be exchanged; give an example where this can not be done. If at all possible, use practical examples from this book.

If you have such choice, there are many factors that can influence your decision.

Programming language: C versus Fortran If your loop describes the (i, j) indices of a two-dimensional array, it is often best to let the i-index be in the inner loop for Fortran, and the j-index inner for C.

Exercise 6.4. Can you come up with at least two reasons why this is possibly better for performance?

However, this is not a hard-and-fast rule. It can depend on the size of the loops, and other factors. For instance, in the matrix-vector product, changing the loop ordering changes how the input and output vectors are used.

Parallelism model If you want to parallelize your loops with *OpenMP*, you generally want the outer loop to be larger than the inner. Having a very short outer loop is definitely bad. A short inner loop can also often be *vectorized by the compiler*. On the other hand, if you are targeting a *GPU*, you want the large loop to be the inner one. The unit of parallel work should not have branches or loops.

Other effects of loop ordering in OpenMP are discussed in *Parallel Programming book, section* 19.6.2.

6.7 Loop tiling

In some cases performance can be increased by breaking up a loop into two nested loops, an outer one for the blocks in the iteration space, and an inner one that goes through the block. This is known as *loop tiling*: the (short) inner loop is a tile, many consecutive instances of which form the iteration space.

For instance

```
for (i=0; i<n; i++)
  ...
```

becomes

```
bs = ...       /* the blocksize */
nblocks = n/bs /* assume that n is a multiple of bs */
for (b=0; b<nblocks; b++)
  for (i=b*bs,j=0; j<bs; i++,j++)
    ...
```

For a single loop this may not make any difference, but given the right context it may. For instance, if an array is repeatedly used, but it is too large to fit into cache:

```
for (n=0; n<10; n++)
  for (i=0; i<100000; i++)
    ... = ...x[i] ...
```

then loop tiling may lead to a situation where the array is divided into blocks that will fit in cache:

```
bs = ... /* the blocksize */
for (b=0; b<100000/bs; b++)
  for (n=0; n<10; n++)
    for (i=b*bs; i<(b+1)*bs; i++)
      ... = ...x[i] ...
```

For this reason, loop tiling is also known as *cache blocking*. The block size depends on how much data is accessed in the loop body; ideally you would try to make data reused in L1 cache, but it is also possible to block for L2 reuse. Of course, L2 reuse will not give as high a performance as L1 reuse.

Exercise 6.5. Analyze this example. When is x brought into cache, when is it reused, and when is it flushed? What is the required cache size in this example? Rewrite this example, using a constant

```
#define L1SIZE 65536
```

For a less trivial example, let's look at *matrix transposition* $A \leftarrow B^t$. Ordinarily you would traverse the input and output matrices:

```
// transposeblock/regular.c
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
    A[i][j] = B[j][i];
```

Using blocking this becomes:

```
// transposeblock/blocked.c
for (int ii=0; ii<N; ii+=blocksize)
  for (int jj=0; jj<N; jj+=blocksize)
    for (int i=ii*blocksize; i<MIN(N,(ii+1)*blocksize); i++)
      for (int j=jj*blocksize; j<MIN(N,(jj+1)*blocksize); j++)
        A[i][j] = B[j][i];
```

Unlike in the example above, each element of the input and output is touched only once, so there is no direct reuse. However, there is reuse of cachelines.

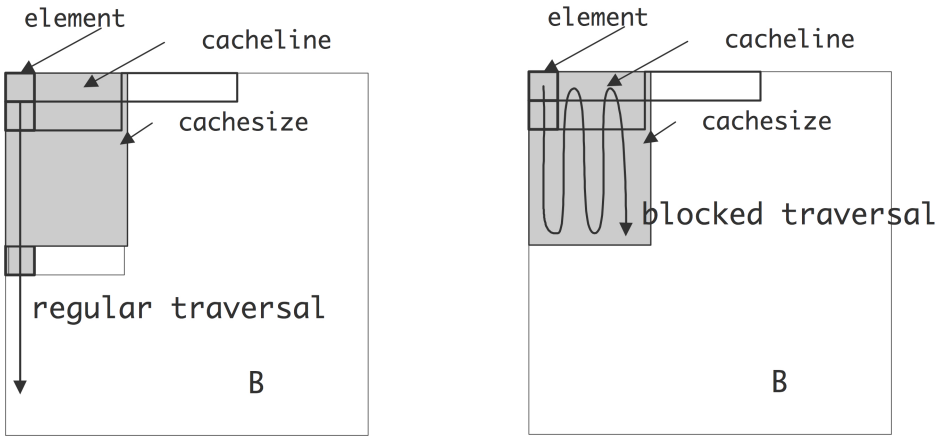


Figure 6.9: Regular and blocked traversal of a matrix.

Figure 6.9 shows how one of the matrices is traversed in a different order from its storage order, for instance columnwise while it is stored by rows. This has the effect that each element load transfers a cacheline, of which only one element is immediately used. In the regular traversal, this streams of cachelines quickly overflows the cache, and there is no reuse. In the blocked traversal, however, only a small number of cachelines is traversed before the next element of these lines is needed. Thus there is reuse of cachelines, or *spatial locality*.

The most important example of attaining performance through blocking is the *matrix!matrix product!tiling*. In section 1.6.2 we looked at the matrix-matrix multiplication, and concluded that little data could be kept in cache. With loop tiling we can improve this situation. For instance, the standard way of writing this product

```
for i=1..n
  for j=1..n
    for k=1..n
      c[i,j] += a[i,k]*b[k,j]
```

can only be optimized to keep `c[i,j]` in register:

```
for i=1..n
  for j=1..n
    s = 0
    for k=1..n
      s += a[i,k]*b[k,j]
```

```
    c[i,j] += s
```

Using loop tiling we can keep parts of `a[i,:]` in cache, assuming that `a` is stored by rows:

```
for kk=1..n/bs
  for i=1..n
    for j=1..n
      s = 0
      for k=(kk-1)*bs+1..kk*bs
        s += a[i,k]*b[k,j]
      c[i,j] += s
```

6.8 Optimization strategies

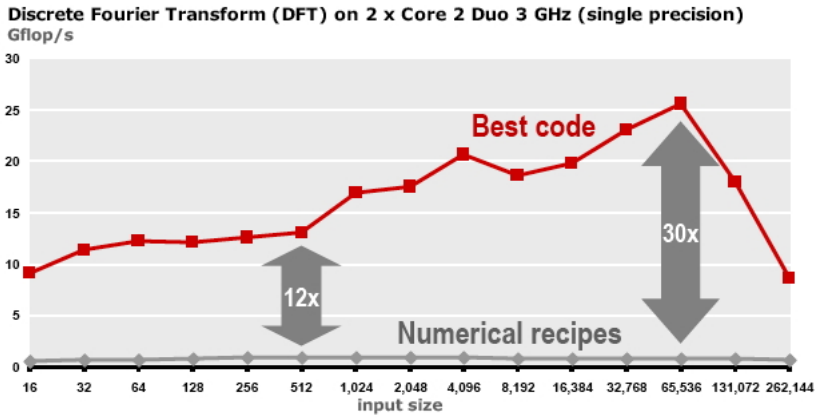


Figure 6.10: Performance of naive and optimized implementations of the Discrete Fourier Transform.

Figures 6.10 and 6.11 show that there can be wide discrepancy between the performance of naive implementations of an operation (sometimes called the 'reference implementation'), and optimized implementations. Unfortunately, optimized implementations are not simple to find. For one, since they rely on blocking, their loop nests are double the normal depth: the matrix-matrix multiplication becomes a six-deep loop. Then, the optimal block size is dependent on factors like the target architecture.

We make the following observations:

- Compilers are not able to extract anywhere close to optimal performance[1] .
- There are *autotuning* projects for automatic generation of implementations that are tuned to the architecture. This approach can be moderately to very successful. Some of the best known of these projects are Atlas [194] for Blas kernels, and Spiral [166] for transforms.

1. Presenting a compiler with the reference implementation may still lead to high performance, since some compilers are trained to recognize this operation. They will then forego translation and simply replace it by an optimized variant.

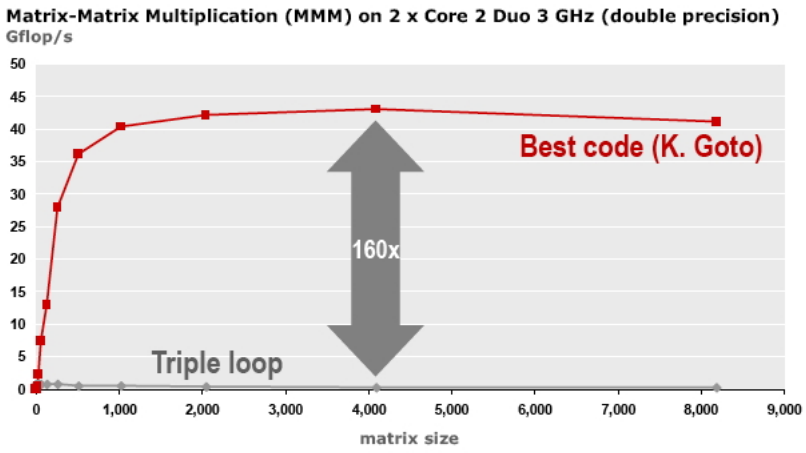


Figure 6.11: Performance of naive and optimized implementations of the matrix-matrix product.

6.9 Cache aware and cache oblivious programming

Unlike registers and main memory, both of which can be addressed in (assembly) code, use of caches is implicit. There is no way a programmer can load data explicitly to a certain cache, even in assembly language.

However, it is possible to code in a 'cache aware' manner. Suppose a piece of code repeatedly operates on an amount of data that is less than the cache size. We can assume that the first time the data is accessed, it is brought into cache; the next time it is accessed it will already be in cache. On the other hand, if the amount of data is more than the cache size[2] , it will partly or fully be flushed out of cache in the process of accessing it.

We can experimentally demonstrate this phenomenon. With a very accurate counter, the code fragment

```
for (x=0; x<NX; x++)
  for (i=0; i<N; i++)
    a[i] = sqrt(a[i]);
```

will take time linear in `N` up to the point where `a` fills the cache. An easier way to picture this is to compute a normalized time, essentially a time per execution of the inner loop:

```
t = time();
for (x=0; x<NX; x++)
  for (i=0; i<N; i++)
    a[i] = sqrt(a[i]);
t = time()-t;
t_normalized = t/(N*NX);
```

The normalized time will be constant until the array `a` fills the cache, then increase and eventually level off again. (See section 6.4.1 for an elaborate discussion.)

2. We are conveniently ignoring matters of set-associativity here, and basically assuming a fully associative cache.

The explanation is that, as long as `a[0]...a[N-1]` fit in L1 cache, the inner loop will use data from the L1 cache. Speed of access is then determined by the latency and bandwidth of the L1 cache. As the amount of data grows beyond the L1 cache size, some or all of the data will be flushed from the L1, and performance will be determined by the characteristics of the L2 cache. Letting the amount of data grow even further, performance will again drop to a linear behavior determined by the bandwidth from main memory.

If you know the cache size, it is possible in cases such as above to arrange the algorithm to use the cache optimally. However, the cache size is different per processor, so this makes your code not portable, or at least its high performance is not portable. Also, blocking for multiple levels of cache is complicated. For these reasons, some people advocate *cache oblivious programming* [68].

Cache oblivious programming can be described as a way of programming that automatically uses all levels of the *cache hierarchy* without knowing the specific sizes. This is typically done by using a *divide-and-conquer* strategy, that is, recursive subdivision of a problem.

A simple example of cache oblivious programming is the *matrix transposition* operation $B \leftarrow A^t$. First we observe that each element of either matrix is accessed once, so the only reuse is in the utilization of cache lines. If both matrices are stored by rows and we traverse B by rows, then A is traversed by columns, and for each element accessed one cacheline is loaded. If the number of rows times the number of elements per cacheline is more than the cachesize, lines will be evicted before they can be reused.

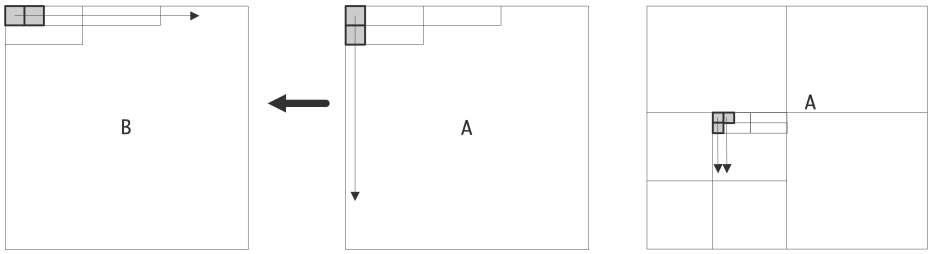


Figure 6.12: Matrix transpose operation, with simple and recursive traversal of the source matrix.

In a cache oblivious implementation we divide A and B as 2×2 block matrices, and recursively compute $B_{11} \leftarrow A_{11}^t$, $B_{12} \leftarrow A_{21}^t$, et cetera; see figure 6.12. At some point in the recursion, blocks A_{ij} will now be small enough that they fit in cache, and the cachelines of A will be fully used. Hence, this algorithm improves on the simple one by a factor approximately equal to the cacheline size.

The cache oblivious strategy can often yield improvement, but it is not necessarily optimal. In the *matrix-matrix product* it improves on the naive algorithm, but it is not as good as an algorithm that is explicitly designed to make optimal use of caches; see section 7.4.1.

See section 7.8.4 for a discussion of such techniques in stencil computations.

6.10 Case study: Matrix-vector product

Let us consider in some detail the *matrix-vector product*

$$\forall_{i,j} : y_i \leftarrow a_{ij} \cdot x_j$$

This involves $2n^2$ operations on $n^2 + 2n$ data items, so reuse is $O(1)$: memory accesses and operations are of the same order. However, we note that there is a double loop involved, and the x, y vectors have only a single index, so each element in them is used multiple times.

Exploiting this theoretical reuse is not trivial. In

```
/* variant 1 */
for (i)
  for (j)
    y[i] = y[i] + a[i][j] * x[j];
```

the element `y[i]` seems to be reused. However, the statement as given here would write `y[i]` to memory in every inner iteration, and we have to write the loop as

```
/* variant 2 */
for (i) {
  s = 0;
  for (j)
    s = s + a[i][j] * x[j];
  y[i] = s;
}
```

to ensure reuse. This variant uses $2n^2$ loads and n stores. This optimization is likely to be done by the compiler.

This code fragment only exploits the reuse of `y` explicitly. If the cache is too small to hold the whole vector `x` plus a column of `a`, each element of `x` is still repeatedly loaded in every outer iteration. Reversing the loops as

```
/* variant 3 */
for (j)
  for (i)
    y[i] = y[i] + a[i][j] * x[j];
```

exposes the reuse of `x`, especially if we write this as

```
/* variant 3 */
for (j) {
  t = x[j];
  for (i)
    y[i] = y[i] + a[i][j] * t;
}
```

but now `y` is no longer reused. Moreover, we now have $2n^2+n$ loads, comparable to variant 2, but n^2 stores, which is of a higher order.

It is possible to get reuse both of x and y, but this requires more sophisticated programming. The key here is to split the loops into blocks. For instance:

```
for (i=0; i<M; i+=2) {
  s1 = s2 = 0;
  for (j) {
    s1 = s1 + a[i][j] * x[j];
```

```
    s2 = s2 + a[i+1][j] * x[j];
  }
  y[i] = s1; y[i+1] = s2;
}
```

This is also called *loop unrolling*, or *strip mining*. The amount by which you unroll loops is determined by the number of available registers.

Chapter 7

High performance linear algebra

In this section we will discuss a number of issues pertaining to linear algebra on parallel computers. We will take a realistic view of this topic, assuming that the number of processors is finite, and that the problem data is always large, relative to the number of processors. We will also pay attention to the physical aspects of the communication network between the processors.

We will analyze various linear algebra operations, including iterative methods for the solution of linear systems of equation (if we sometimes just refer to 'iterative methods', the qualification to systems of linear equations is implicit), and their behavior in the presence of a network with finite bandwidth and finite connectivity.

7.1 Collective operations

Collective operations are an artifact of distributed memory parallelism; see section 2.4 for a discussion. Mathematically they describe extremely simple operations such as summing an array of numbers. However, since these numbers can be on different processors, each with their own memory space, this operation becomes computationally non-trivial and worthy of study.

Collective operations play an important part in linear algebra operations. In fact, the scalability of even simple operations such as a matrix-vector product can depend on the cost of these collectives as you will see below. We start with a short discussion of the essential ideas behind collectives; see [29] for details. We will then see an important application in section 7.2, and various other places.

In computing the cost of a collective operation, three architectural constants are enough to give lower bounds: α, the *latency* of sending a single message, β, the inverse of the *bandwidth* for sending data (see section 1.3.2), and γ, the inverse of the *computation rate*, the time for performing an arithmetic operation. Sending n data items then takes time $\alpha+\beta n$. We further assume that a processor can only send one message at a time. We make no assumptions about the connectivity of the processors (see section 2.7 for this); thus, the lower bounds derived here will hold for a wide range of architectures.

The main implication of the architectural model above is that the number of active processors can only double in each step of an algorithm. For instance, to do a broadcast, first processor 0 sends to 1, then 0 and 1 can send to 2 and 3, then 0–3 send to 4–7, et cetera. This cascade of messages is called a *minimum*

spanning tree of the processor network, and it follows that any collective algorithm has at least $\alpha \log_2 p$ cost associated with the accumulated latencies.

7.1.1 Broadcast

In a *broadcast* operation, a single processor has n data elements that it needs to send to all others: the other processors need a full copy of all n elements. By the above doubling argument, we conclude that a broadcast to p processors takes time at least $\lceil \log_2 p \rceil$ steps with a total latency of $\lceil \log_2 p \rceil \alpha$. Since n elements are sent, this adds at least a time $n\beta$ for all elements to leave the sending processor, giving a total cost lower bound of

$$\lceil \log_2 p \rceil \alpha + n\beta.$$

We can illustrate the spanning tree method as follows:

	$t=1$	$t=2$	$t=3$
p_0	$x_0\downarrow, x_1\downarrow, x_2\downarrow, x_3\downarrow$	$x_0\downarrow, x_1\downarrow, x_2\downarrow, x_3\downarrow$	x_0, x_1, x_2, x_3
p_1		$x_0\downarrow, x_1\downarrow, x_2\downarrow, x_3\downarrow$	x_0, x_1, x_2, x_3
p_2			x_0, x_1, x_2, x_3
p_3			x_0, x_1, x_2, x_3

(On $t=1$, p_0 sends to p_1; on $t=2$ p_0, p_1 send to p_2, p_3.) This algorithm has the correct $\log_2 p \cdot \alpha$ term, but processor 0 repeatedly sends the whole vector, so the bandwidth cost is $\log_2 p \cdot n\beta$. If n is small, the latency cost dominates, so we may characterize this as a *short vector collective operation*

The following algorithm implements the broadcast as a combination of a scatter and a *bucket brigade algorithm*. First the scatter:

	$t=0$	$t=1$	$t=2$	$t=3$
p_0	$x_0\downarrow, x_1, x_2, x_3$	$x_0, x_1\downarrow, x_2, x_3$	$x_0, x_1, x_2\downarrow, x_3$	$x_0, x_1, x_2, x_3\downarrow$
p_1		x_1		
p_2			x_2	
p_3				x_3

This involves $p-1$ messages of size N/p, for a total time of

$$T_{\text{scatter}}(N,P) = (p-1)\alpha + (p-1)\cdot\frac{N}{p}\cdot\beta.$$

Then the bucket brigade has each processor active in every step, accepting a partial message (except in the first step), and passing it on to the next processor.

	$t=0$	$t=1$	*etcetera*
p_0	$x_0\downarrow$	$x_0 \quad x_3\downarrow$	$x_0, \quad x_2, x_3$
p_1	$x_1\downarrow$	$x_0\downarrow, x_1$	$x_0, x_1, \quad x_3$
p_2	$x_2\downarrow$	$x_1\downarrow, x_2$	x_0, x_1, x_2
p_3	$x_3\downarrow$	$x_2\downarrow, x_3$	x_1, x_2, x_3

Each partial message gets sent $p-1$ times, so this stage also has a complexity of

$$T_{\text{bucket}}(N,P) = (p-1)\alpha + (p-1)\cdot\frac{N}{p}\cdot beta.$$

The complexity now becomes

$$2(p-1)\alpha + 2\beta n(p-1)/p$$

which is not optimal in latency, but is a better algorithm if n is large, making this a *long vector collective operation.*

7.1.2 Reduction

In a *reduction* operation, each processor has n data elements, and one processor needs to combine them elementwise, for instance computing n sums or products.

By running the broadcast backwards in time, we see that a reduction operation has the same lower bound on the communication of $\lceil \log_2 p \rceil \alpha + n\beta$. A reduction operation also involves computation, which would take a total time of $(p-1)\gamma n$ sequentially: each of n items gets reduced over p processors. Since these operations can potentially be parallelized, the lower bound on the computation is $\frac{p-1}{p}\gamma n$, giving a total of

$$\lceil \log_2 p \rceil \alpha + n\beta + \frac{p-1}{p}\gamma n.$$

We illustrate the spanning tree algorithm, using the notation $x_i^{(j)}$ for the data item i that was originally on processor j, and $x_i^{(j:k)}$ for the sum of the items i of processors $j \ldots k$.

	$t=1$	$t=2$	$t=3$
p_0	$x_0^{(0)}, x_1^{(0)}, x_2^{(0)}, x_3^{(0)}$	$x_0^{(0:1)}, x_1^{(0:1)}, x_2^{(0:1)}, x_3^{(0:1)}$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$
p_1	$x_0^{(1)}\uparrow, x_1^{(1)}\uparrow, x_2^{(1)}\uparrow, x_3^{(1)}\uparrow$		
p_2	$x_0^{(2)}, x_1^{(2)}, x_2^{(2)}, x_3^{(2)}$	$x_0^{(2:3)}\uparrow, x_1^{(2:3)}\uparrow, x_2^{(2:3)}\uparrow, x_3^{(2:3)}\uparrow$	
p_3	$x_0^{(3)}\uparrow, x_1^{(3)}\uparrow, x_2^{(3)}\uparrow, x_3^{(3)}\uparrow$		

On time $t=1$ processors p_0, p_2 receive from p_1, p_3, and on $t=2$ p_0 receives from p_2.

As with the broadcast above, this algorithm does not achieve the lower bound; instead it has a complexity

$$\lceil \log_2 p \rceil (\alpha + n\beta + \frac{p-1}{p}\gamma n).$$

For short vectors the α term dominates, so this algorithm is sufficient. For long vectors one can, as above, use other algorithms [29].

A *long vector reduction* can be done using a bucket brigade followed by a gather. The complexity is as above, except that the bucket brigade performs partial reductions, for a time of $\gamma(p-1)N/p$. The gather performs no further operations.

7.1.3 Allreduce

An *allreduce* operation computes the same elementwise reduction of n elements on each processor, but leaves the result on each processor, rather than just on the root of the spanning tree. This could be implemented as a reduction followed by a broadcast, but more clever algorithms exist.

	$t=1$	$t=2$	$t=3$
p_0	$x_0^{(0)}\downarrow, x_1^{(0)}\downarrow, x_2^{(0)}\downarrow, x_3^{(0)}\downarrow$	$x_0^{(0:1)}\downarrow\downarrow, x_1^{(0:1)}\downarrow\downarrow, x_2^{(0:1)}\downarrow\downarrow, x_3^{(0:1)}\downarrow\downarrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$
p_1	$x_0^{(1)}\uparrow, x_1^{(1)}\uparrow, x_2^{(1)}\uparrow, x_3^{(1)}\uparrow$	$x_0^{(0:1)}\downarrow\downarrow, x_1^{(0:1)}\downarrow\downarrow, x_2^{(0:1)}\downarrow\downarrow, x_3^{(0:1)}\downarrow\downarrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$
p_2	$x_0^{(2)}\downarrow, x_1^{(2)}\downarrow, x_2^{(2)}\downarrow, x_3^{(2)}\downarrow$	$x_0^{(2:3)}\uparrow\uparrow, x_1^{(2:3)}\uparrow\uparrow, x_2^{(2:3)}\uparrow\uparrow, x_3^{(2:3)}\uparrow\uparrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$
p_3	$x_0^{(3)}\uparrow, x_1^{(3)}\uparrow, x_2^{(3)}\uparrow, x_3^{(3)}\uparrow$	$x_0^{(2:3)}\uparrow\uparrow, x_1^{(2:3)}\uparrow\uparrow, x_2^{(2:3)}\uparrow\uparrow, x_3^{(2:3)}\uparrow\uparrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$

The lower bound on the cost of an allreduce is, somewhat remarkably, almost the same as of a simple reduction: since in a reduction not all processors are active at the same time, we assume that the extra work can be spread out perfectly. This means that the lower bound on the latency and computation stays the same.

Since every processor originates a minimum spanning tree, we have $\log_2 p\alpha$ latency. For the bandwidth we reason as follows: in order for the communication to be perfectly parallelized, $\frac{p-1}{p}n$ items have to arrive at, and leave each processor. Thus we have a total time of

$$\lceil \log_2 p \rceil \alpha + 2\frac{p-1}{p}n\beta + \frac{p-1}{p}n\gamma.$$

The allreduce is an important operation in iterative methods for linear systems; see for instance section 5.5.9. There, the reduction is typically applied to a single scalar, meaning that latency is relatively important.

7.1.4 Allgather

In a *gather* operation on n elements, each processor has n/p elements, and one processor collects them all, without combining them as in a reduction. The *allgather* computes the same gather, but leaves the result on all processors. You will see two important applications of this in the dense matrix-vector product (section 7.2.3.3), and the setup of the sparse matrix-vector product (section 7.5.6).

Again we assume that gathers with multiple targets are active simultaneously. Since every processor originates a minimum spanning tree, we have $\log_2 p\alpha$ latency; since each processor receives n/p elements from $p-1$ processors, there is $(p-1)\times(n/p)\beta$ bandwidth cost. The total cost for constructing a length n vector by allgather is then

$$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p}n\beta.$$

We illustrate this:

	$t=1$	$t=2$	$t=3$
p_0	$x_0\downarrow$	$x_0x_1\downarrow$	$x_0x_1x_2x_3$
p_1	$x_1\uparrow$	$x_0x_1\downarrow$	$x_0x_1x_2x_3$
p_2	$x_2\downarrow$	$x_2x_3\uparrow$	$x_0x_1x_2x_3$
p_3	$x_3\uparrow$	$x_2x_3\uparrow$	$x_0x_1x_2x_3$

At time $t=1$, there is an exchange between neighbors p_0,p_1 and likewise p_2,p_3; at $t=2$ there is an exchange over distance two between p_0,p_2 and likewise p_1,p_3.

7.1.5 Reduce-scatter

In a *reduce-scatter* operation, each processor has n elements, and an n-way reduction is done on them. Unlike in the reduce or allreduce, the result is then broken up, and distributed as in a scatter operation.

Formally, processor i has an item $x_i^{(i)}$, and it needs $\sum_j x_i^{(j)}$. We could implement this by doing a size p reduction, collecting the vector $(\sum_i x_0^{(i)}, \sum_i x_1^{(i)}, \ldots)$ on one processor, and scattering the results. However it is possible to combine these operations in a so-called *bidirectional exchange* algorithm:

	$t=1$	$t=2$	$t=3$
p_0	$x_0^{(0)}, x_1^{(0)}, x_2^{(0)}\downarrow, x_3^{(0)}\downarrow$	$x_0^{(0:2:2)}, x_1^{(0:2:2)}\downarrow$	$x_0^{(0:3)}$
p_1	$x_0^{(1)}, x_1^{(1)}, x_2^{(1)}\downarrow, x_3^{(1)}\downarrow$	$x_0^{(1:3:2)}\uparrow, x_1^{(1:3:2)}$	$x_1^{(0:3)}$
p_2	$x_0^{(2)}\uparrow, x_1^{(2)}\uparrow, x_2^{(2)}, x_3^{(2)}$	$x_2^{(0:2:2)}, x_3^{(0:2:2)}\downarrow$	$x_2^{(0:3)}$
p_3	$x_0^{(3)}\uparrow, x_1^{(3)}\uparrow, x_2^{(3)}, x_3^{(3)}$	$x_0^{(1:3:2)}\uparrow, x_1^{(1:3:2)}$	$x_3^{(0:3)}$

The reduce-scatter can be considered as a allgather run in reverse, with arithmetic added, so the cost is

$$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p} n(\beta+\gamma).$$

7.2 Parallel dense matrix-vector product

In this section we will go into great detail into the performance, and in particular the scalability, of the parallel dense matrix-vector product. First we will consider a simple case, and discuss the parallelism aspects in some amount of detail.

7.2.1 Implementing the block-row case

In designing a parallel version of an algorithm, one often proceeds by making a *data decomposition* of the objects involved. In the case of a matrix-vector operations such as the product $y = Ax$, we have the choice of starting with a vector decomposition, and exploring its ramifications on how the matrix can be decomposed, or rather to start with the matrix, and deriving the vector decomposition from it. In this case, it seems natural to start with decomposing the matrix rather than the vector, since it will be most likely of larger computational significance. We now have two choices:

1. We make a one-dimensional decomposition of the matrix, splitting it in block rows or block columns, and assigning each of these – or groups of them – to a processor.
2. Alternatively, we can make a two-dimensional decomposition, assigning to each processor one or more general submatrices.

We start by considering the decomposition in block rows. Consider a processor p and the set I_p of indices of rows that it owns[1], and let $i \in I_p$ be a row that is assigned to this processor. The elements in row i are used in the operation

$$y_i = \sum_j a_{ij} x_j$$

We now reason:

- If processor p has all x_j values, the matrix-vector product can trivially be executed, and upon completion, the processor has the correct values y_j for $j \in I_p$.
- This means that every processor needs to have a copy of x, which is wasteful. Also it raises the question of data integrity: you need to make sure that each processor has the correct value of x.
- In certain practical applications (for instance iterative methods, as you have seen before), the output of the matrix-vector product is, directly or indirectly, the input for a next matrix-vector operation. This is certainly the case for the power method which computes $x, Ax, A^2x, \ldots$. Since our operation started with each processor having the whole of x, but ended with it owning only the local part of Ax, we have a mismatch.
- Maybe it is better to assume that each processor, at the start of the operation, has only the local part of x, that is, those x_i where $i \in I_p$, so that the start state and end state of the algorithm are the same. This means we have to change the algorithm to include some communication that allows each processor to obtain those values x_i where $i \notin I_p$.

Exercise 7.1. Go through a similar reasoning for the case where the matrix is decomposed in block columns. Describe the parallel algorithm in detail, like above, but without giving pseudo code.

Let us now look at the communication in detail: we will consider a fixed processor p and consider the operations it performs and the communication that necessitates. According to the above analysis, in executing the statement $y_i = \sum_j a_{ij} x_j$ we have to be aware what processor the j values 'belong to'. To acknowledge this, we write

$$y_i = \sum_{j \in I_p} a_{ij} x_j + \sum_{j \notin I_p} a_{ij} x_j \tag{7.1}$$

1. For ease of exposition we will let I_p be a contiguous range of indices, but any general subset is allowed.

Input: Processor number p; the elements x_i with $i \in I_p$; matrix elements A_{ij} with $i \in I_p$.

Output: The elements y_i with $i \in I_p$

for $i \in I_p$ **do**
  $s \leftarrow 0$;
  for $j \in I_p$ **do**
    $s \leftarrow s + a_{ij}x_j$
  for $j \notin I_p$ **do**
    send x_j from the processor that owns it to the current one, then;
    $s \leftarrow s + a_{ij}x_j$
  $y_i \leftarrow s$

Procedure Naive Parallel MVP($A, x_{local}, y_{local}, p$)

Figure 7.1: A naïvely coded parallel matrix-vector product.

If $j \in I_p$, the instruction $y_i \leftarrow y_i + a_{aij}x_j$ involves only quantities that are already local to the processor. Let us therefore concentrate on the case $j \notin I_p$. It would be nice if we could just write the statement

```
y(i) = y(i) + a(i,j)*x(j)
```

and some lower layer would automatically transfer `x(j)` from whatever processor it is stored on to a local register. (The PGAS languages of section 2.6.5 aim to do this, but their efficiency is far from guaranteed.) An implementation, based on this optimistic view of parallelism, is given in figure 7.1.

The immediate problem with such a 'local' approach is that too much communication will take place.

- If the matrix A is dense, the element x_j is necessary once for each row $i \in I_p$, and it will thus be fetched once for every row $i \in I_p$.
- For each processor $q \neq p$, there will be (large) number of elements x_j with $j \in I_q$ that need to be transferred from processor q to p. Doing this in separate messages, rather than one bulk transfer, is very wasteful.

With shared memory these issues are not much of a problem, but in the context of distributed memory it is better to take a *buffering* approach.

Instead of communicating individual elements of x, we use a local buffer B_{pq} for each processor $q \neq p$ where we collect the elements from q that are needed to perform the product on p. (See figure 7.2 for an illustration.) The parallel algorithm is given in figure 7.3.

In addition to preventing an element from being fetched more than once, this also combines many small messages into one large message, which is usually more efficient; recall our discussion of bandwidth and latency in section 2.7.8.

Exercise 7.2. Give pseudocode for the matrix-vector product using nonblocking operations (section 2.6.3.6)

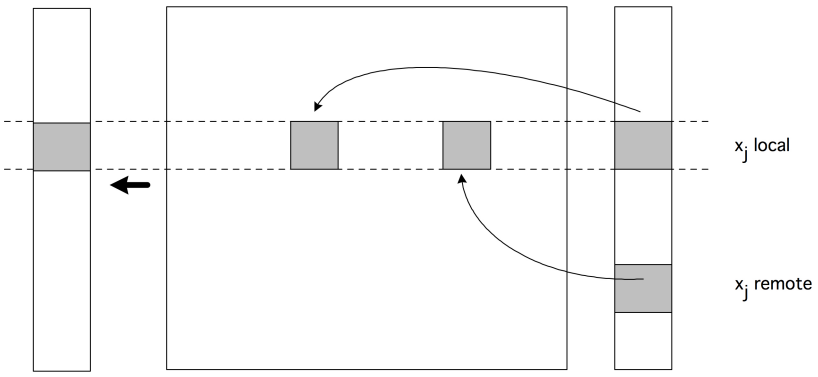


Figure 7.2: The parallel matrix-vector product with a blockrow distribution.

Input: Processor number p; the elements x_i with $i \in I_p$; matrix elements A_{ij} with $i \in I_p$.

Output: The elements y_i with $i \in I_p$

for $q \neq p$ **do**
Send elements of x from processor q to p, receive in buffer B_{pq}.

$y_{local} \leftarrow Ax_{local}$
for $q \neq p$ **do**
$y_{local} \leftarrow y_{local} + A_{pq}B_q$
Procedure Parallel MVP($A, x_{local}, y_{local}, p$)

Figure 7.3: A buffered implementation of the parallel matrix-vector product.

Above we said that having a copy of the whole of x on each processor was wasteful in space. The implicit argument here is that, in general, we do not want local storage to be function of the number of processors: ideally it should be only a function of the local data. (This is related to weak scaling; section 2.2.5.)

You see that, because of communication considerations, we have actually decided that it is unavoidable, or at least preferable, for each processor to store the whole input vector. Such trade-offs between space and time efficiency are fairly common in parallel programming. For the dense matrix-vector product we can actually defend this overhead, since the vector storage is of lower order than the matrix storage, so our over-allocation is small by ratio. Below (section 7.5), we will see that for the sparse matrix-vector product the overhead can be much less.

It is easy to see that the parallel dense matrix-vector product, as described above, has perfect speedup *if we are allowed to ignore the time for communication*. In the next couple of sections you will see that the block row implementation above is not optimal if we take communication into account. For scalability we need a two-dimensional decomposition. We start with a discussion of collectives.

7.2.2 The block-column case

We can also distribute the matrix by block columns. If you have studied the block-row case, this is not much different:

- We start out again with each process storing a distinct set of columns;
- and each process starts out storing the corresponding part of the input;
- after completion of the product operation, each process stores its section of the output.

Exercise 7.3. Sketch the algorithm in pseudo-code. Where does the communication now happen relative to the computation? What collective operation are you using? (Hint: each process needs only part of the output, not the whole vector.)

7.2.3 Scalability of the dense matrix-vector product

In this section, we will give a full analysis of the parallel computation of $y \leftarrow Ax$, where $x, y \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n\times n}$. We will assume that p processes will be used, but we make no assumptions on their connectivity. We will see that the way the matrix is distributed makes a big difference for the scaling of the algorithm; for the original research see [99, 173, 181], and see section 2.2.5 for the definitions of the various forms of scaling.

7.2.3.1 Matrix-vector product, partitioning by rows

Partition

$$A \to \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{pmatrix} \quad x \to \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{pmatrix}, \quad \text{and} \quad y \to \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{p-1} \end{pmatrix},$$

where $A_i \in \mathbb{R}^{m_i\times n}$ and $x_i, y_i \in \mathbb{R}^{m_i}$ with $\sum_{i=0}^{p-1} m_i = n$ and $m_i \approx n/p$. We will start by assuming that A_i, x_i, and y_i are originally assigned to P_i.

The computation is characterized by the fact that each processor needs the whole vector x, but owns only an n/p fraction of it. Thus, we execute an *allgather* of x. After this, the processor can execute the local product $y_i \leftarrow A_i x$; no further communication is needed after that.

An algorithm with cost computation for $y = Ax$ in parallel is then given by

Step	Cost (lower bound)
Allgather x_i so that x is available on all nodes	$\lceil \log_2(p) \rceil \alpha + \frac{p-1}{p} n\beta \approx \log_2(p)\alpha + n\beta$
Locally compute $y_i = A_i x$	$\approx 2\frac{n^2}{p}\gamma$

Cost analysis The total cost of the algorithm is given by, approximately,

$$T_p(n) = T_p^{\text{1D-row}}(n) = 2\frac{n^2}{p}\gamma + \underbrace{\log_2(p)\alpha + n\beta}_{\text{Overhead}}.$$

Since the sequential cost is $T_1(n) = 2n^2\gamma$, the speedup is given by

$$S_p^{\text{1D-row}}(n) = \frac{T_1(n)}{T_p^{\text{1D-row}}(n)} = \frac{2n^2\gamma}{2\frac{n^2}{p}\gamma + \log_2(p)\alpha + n\beta} = \frac{p}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}}$$

and the parallel efficiency by

$$E_p^{\text{1D-row}}(n) = \frac{S_p^{\text{1D-row}}(n)}{p} = \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}}.$$

7.2.3.1.1 **An optimist's view** Now, if one fixes p and lets n get large,

$$\lim_{n\to\infty} E_p(n) = \lim_{n\to\infty} \left[\frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}} \right] = 1.$$

Thus, if one can make the problem large enough, eventually the parallel efficiency is nearly perfect. However, this assumes unlimited memory, so this analysis is not practical.

7.2.3.1.2 **A pessimist's view** In a *strong scalability* analysis, one fixes n and lets p get large, to get

$$\lim_{p\to\infty} E_p(n) = \lim_{p\to\infty} \left[\frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}} \right] \sim \frac{1}{p} \downarrow 0.$$

Thus, eventually the parallel efficiency becomes nearly nonexistent.

7.2.3.1.3 **A realist's view** In a more realistic view we increase the number of processors with the amount of data. This is called *weak scalability*, and it makes the amount of memory that is available to store the problem scale linearly with p.

Let M equal the number of floating point numbers that can be stored in a single node's memory. Then the aggregate memory is given by Mp. Let $n_{\max}(p)$ equal the largest problem size that can be stored in the aggregate memory of p nodes. Then, if *all* memory can be used for the matrix,

$$(n_{\max}(p))^2 = Mp \quad \text{or} \quad n_{\max}(p) = \sqrt{Mp}.$$

The question now becomes what the parallel efficiency for the largest problem that can be stored on p nodes:

$$\begin{aligned} E_p^{\text{1D-row}}(n_{\max}(p)) &= \frac{1}{1+\frac{p\log_2(p)}{2(n_{\max}(p))^2}\frac{\alpha}{\gamma}+\frac{p}{2n_{\max}(p)}\frac{\beta}{\gamma}} \\ &= \frac{1}{1+\frac{\log_2(p)}{2M}\frac{\alpha}{\gamma}+\frac{\sqrt{p}}{2\sqrt{M}}\frac{\beta}{\gamma}}. \end{aligned}$$

Now, if one analyzes what happens when the number of nodes becomes large, one finds that

$$\lim_{p\rightarrow\infty} E_p(n_{\max}(p)) = \lim_{p\rightarrow\infty} \left[\frac{1}{1+\frac{\log_2(p)}{2M}\frac{\alpha}{\gamma}+\frac{\sqrt{p}}{2\sqrt{M}}\frac{\beta}{\gamma}} \right] \sim \frac{1}{\sqrt{p}} \downarrow 0.$$

Thus, this parallel algorithm for matrix-vector multiplication does not scale either.

If you take a close look at this expression for efficiency, you'll see that the main problem is the $1/\sqrt{p}$ part of the expression. This terms involves a factor β, and if you follow the derivation backward you see that it comes from the time to send data between the processors. Informally this can be described as saying that the message size is too large to make the problem scalable. In fact, the message size is constant n, regardless the number of processors, while for scalability it probably needs to go down.

Alternatively, a realist realizes that there is a limited amount of time, $T_{\max}$, to get a computation done. Under the best of circumstances, that is, with zero communication overhead, the largest problem that we can solve in time $T_{\max}$ is given by

$$T_p(n_{\max}(p)) = 2\frac{(n_{\max}(p))^2}{p}\gamma = T_{\max}.$$

Thus

$$(n_{\max}(p))^2 = \frac{T_{\max}p}{2\gamma} \quad \text{or} \quad n_{\max}(p) = \frac{\sqrt{T_{\max}}\sqrt{p}}{\sqrt{2\gamma}}.$$

Then the parallel efficiency that is attained by the algorithm for the largest problem that can be solved in time $T_{\max}$ is given by

$$E_{p,n_{\max}} = \frac{1}{1+\frac{\log_2 p}{T}\alpha+\sqrt{\frac{p}{T}\frac{\beta}{\gamma}}}$$

and the parallel efficiency as the number of nodes becomes large approaches

$$\lim_{p\to\infty} E_p = \sqrt{\frac{T\gamma}{p\beta}}.$$

Again, efficiency cannot be maintained as the number of processors increases and the execution time is capped.

We can also compute the *iso-efficiency curve* for this operation, that is, the relationship between n, p for which the efficiency stays constant (see section 2.2.5.1). If we simplify the efficiency above as $E(n,p) = \frac{2\gamma}{\beta}\frac{n}{p}$, then $E \equiv c$ is equivalent to $n = O(p)$ and therefore

$$M = O\left(\frac{n^2}{p}\right) = O(p).$$

Thus, in order to maintain efficiency we need to increase the memory per processor pretty quickly. This makes sense, since that downplays the importance of the communication.

7.2.3.2 Matrix-vector product, partitioning by columns

Partition

$$A \to \left(A_0, A_1, \ldots, A_{p-1}\right) \quad x \to \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{pmatrix}, \quad \text{and} \quad y \to \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{p-1} \end{pmatrix},$$

where $A_j \in \mathbb{R}^{n\times n_j}$ and $x_j, y_j \in \mathbb{R}^{n_j}$ with $\sum_{j=0}^{p-1} n_j = n$ and $n_j \approx n/p$.

We will start by assuming that A_j, x_j, and y_j are originally assigned to P_j (but now A_i is a block of columns). In this algorithm by columns, processor i can compute the length n vector $A_i x_i$ without prior communication. These partial results then have to be added together

$$y \leftarrow \sum_i A_i x_i$$

in a *reduce-scatter* operation: each processor i scatters a part $(A_i x_i)_j$ of its result to processor j. The receiving processors then perform a reduction, adding all these fragments:

$$y_j = \sum_i (A_i x_i)_j.$$

The algorithm with costs is then given by:

Step	Cost (lower bound)
Locally compute $y^{(j)} = A_j x_j$	$\approx 2\frac{n^2}{p}\gamma$
Reduce-scatter the $y^{(j)}$s so that $y_i = \sum_{j=0}^{p-1} y_i^{(j)}$ is on P_i	$\lceil \log_2(p) \rceil \alpha + \frac{p-1}{p} n\beta + \frac{p-1}{p} n\gamma$ $\approx \log_2(p)\alpha + n(\beta+\gamma)$

7.2.3.2.1 **Cost analysis** The total cost of the algorithm is given by, approximately,

$$T_p^{\text{1D-col}}(n) = 2\frac{n^2}{p}\gamma + \underbrace{\log_2(p)\alpha + n(\beta+\gamma)}_{\text{Overhead}}.$$

Notice that this is identical to the cost $T_p^{\text{1D-row}}(n)$, except with β replaced by $(\beta+\gamma)$. It is not hard to see that the conclusions about scalability are the same.

7.2.3.3 Two-dimensional partitioning

Clearly, a one-dimensional partitioning, whether by rows or columns, is a badly scaling solution. The choice left to us is to use a two-dimensional partitioning. Here we order the processes in a two-dimensional grid, whether that makes sense or not in a network sense, and give each process a submatrix that spans less than a full row or column. We will only consider square matrices, but we allow for the process grid not to be perfectly square.

$$A \rightarrow \begin{pmatrix} A_{00} & A_{01} & \cdots & A_{0,c-1} \\ A_{10} & A_{11} & \cdots & A_{1,c-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{r-1,0} & A_{r-1,0} & \cdots & A_{r-1,c-1} \end{pmatrix} \quad x \rightarrow \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{c-1} \end{pmatrix}, \quad \text{and} \quad y \rightarrow \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{r-1} \end{pmatrix},$$

where $A_{ij} \in \mathbb{R}^{n_i \times n_j}$ and $x_i, y_i \in \mathbb{R}^{n_i}$ with $\sum_{i=0}^{p-1} n_i = N$ and $n_i \approx N/\sqrt{P}$.

We will view the processes as an $r \times c$ mesh, with $P = rc$, and index them as p_{ij}, with $i = 0, \ldots, r-1$ and $j = 0, \ldots, c-1$. Figure 7.4, for a 12×12 matrix on a 3×4 processor grid, illustrates the assignment of data to nodes, where the i, j 'cell' shows the matrix and vector elements owned by p_{ij}.

x_0				x_3				x_6				x_9			
a_{00}	a_{01}	a_{02}	y_0	a_{03}	a_{04}	a_{05}		a_{06}	a_{07}	a_{08}		a_{09}	$a_{0,10}$	$a_{0,11}$	
a_{10}	a_{11}	a_{12}		a_{13}	a_{14}	a_{15}	y_1	a_{16}	a_{17}	a_{18}		a_{19}	$a_{1,10}$	$a_{1,11}$	
a_{20}	a_{21}	a_{22}		a_{23}	a_{24}	a_{25}		a_{26}	a_{27}	a_{28}	y_2	a_{29}	$a_{2,10}$	$a_{2,11}$	
a_{30}	a_{31}	a_{32}		a_{33}	a_{34}	a_{35}		a_{37}	a_{37}	a_{38}		a_{39}	$a_{3,10}$	$a_{3,11}$	y_3
	x_1				x_4				x_7				x_{10}		
a_{40}	a_{41}	a_{42}	y_4	a_{43}	a_{44}	a_{45}		a_{46}	a_{47}	a_{48}		a_{49}	$a_{4,10}$	$a_{4,11}$	
a_{50}	a_{51}	a_{52}		a_{53}	a_{54}	a_{55}	y_5	a_{56}	a_{57}	a_{58}		a_{59}	$a_{5,10}$	$a_{5,11}$	
a_{60}	a_{61}	a_{62}		a_{63}	a_{64}	a_{65}		a_{66}	a_{67}	a_{68}	y_6	a_{69}	$a_{6,10}$	$a_{6,11}$	
a_{70}	a_{71}	a_{72}		a_{73}	a_{74}	a_{75}		a_{77}	a_{77}	a_{78}		a_{79}	$a_{7,10}$	$a_{7,11}$	y_7
		x_2				x_5				x_8				x_{11}	
a_{80}	a_{81}	a_{82}	y_8	a_{83}	a_{84}	a_{85}		a_{86}	a_{87}	a_{88}		a_{89}	$a_{8,10}$	$a_{8,11}$	
a_{90}	a_{91}	a_{92}		a_{93}	a_{94}	a_{95}	y_9	a_{96}	a_{97}	a_{98}		a_{99}	$a_{9,10}$	$a_{9,11}$	
$a_{10,0}$	$a_{10,1}$	$a_{10,2}$		$a_{10,3}$	$a_{10,4}$	$a_{10,5}$		$a_{10,6}$	$a_{10,7}$	$a_{10,8}$	y_{10}	$a_{10,9}$	$a_{10,10}$	$a_{10,11}$	
$a_{11,0}$	$a_{11,1}$	$a_{11,2}$		$a_{11,3}$	$a_{11,4}$	$a_{11,5}$		$a_{11,7}$	$a_{11,7}$	$a_{11,8}$		$a_{11,9}$	$a_{11,10}$	$a_{11,11}$	y_{11}

Figure 7.4: Distribution of matrix and vector elements for a problem of size 12 on a 4×3 processor grid.

In other words, p_{ij} owns the matrix block A_{ij} and parts of x and y. This makes possible the following algorithm[2]:

- Since x_j is distributed over the jth column, the algorithm starts by collecting x_j on each processor p_{ij} by an *allgather* inside the processor columns.
- Each processor p_{ij} then computes $y_{ij} = A_{ij}x_j$. This involves no further communication.
- The result y_i is then collected by gathering together the pieces y_{ij} in each processor row to form y_i, and this is then distributed over the processor row. These two operations are in fact combined to form a *reduce-scatter*.
- If $r = c$, we can transpose the y data over the processors, so that it can function as the input for a subsequent matrix-vector product. If, on the other hand, we are computing A^tAx, then y is now correctly distributed for the A^t product.

7.2.3.3.1 **Algorithm** The algorithm with cost analysis is

Step	Cost (lower bound)
Allgather x_i's within columns	$\lceil \log_2(r) \rceil \alpha + \frac{r-1}{p} n\beta$ $\approx \log_2(r)\alpha + \frac{n}{c}\beta$
Perform local matrix-vector multiply	$\approx 2\frac{n^2}{p}\gamma$
Reduce-scatter y_i's within rows	$\lceil \log_2(c) \rceil \alpha + \frac{c-1}{p} n\beta + \frac{c-1}{p} n\gamma$ $\approx \log_2(c)\alpha + \frac{n}{r}\beta + \frac{n}{r}\gamma$

7.2.3.3.2 **Cost analysis** The total cost of the algorithm is given by, approximately,

$$T_p^{r\times c}(n) = T_p^{r\times c}(n) = 2\frac{n^2}{p}\gamma + \underbrace{\log_2(p)\alpha + \left(\frac{n}{c} + \frac{n}{r}\right)\beta + \frac{n}{r}\gamma}_{\text{Overhead}}.$$

We will now make the simplification that $r = c = \sqrt{p}$ so that

$$T_p^{\sqrt{p}\times\sqrt{p}}(n) = T_p^{\sqrt{p}\times\sqrt{p}}(n) = 2\frac{n^2}{p}\gamma + \underbrace{\log_2(p)\alpha + \frac{n}{\sqrt{p}}(2\beta + \gamma)}_{\text{Overhead}}$$

Since the sequential cost is $T_1(n) = 2n^2\gamma$, the speedup is given by

$$S_p^{\sqrt{p}\times\sqrt{p}}(n) = \frac{T_1(n)}{T_p^{\sqrt{p}\times\sqrt{p}}(n)} = \frac{2n^2\gamma}{2\frac{n^2}{p}\gamma + \frac{n}{\sqrt{p}}(2\beta+\gamma)} = \frac{p}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n}\frac{(2\beta+\gamma)}{\gamma}}$$

2. This figure shows a partitioning of the matrix into contiguous blocks, and the vector distribution seem to be what is necessary to work with this matrix distribution. You could also look at this story the other way: start with a distribution of input and output vector, and then decide what that implies for the matrix distribution. For instance, if you distributed x and y the same way, you would arrive at a different matrix distribution, but otherwise the product algorithm would be much the same; see [55].

and the parallel efficiency by

$$E_p^{\sqrt{p}\times\sqrt{p}}(n) = \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n}\frac{(2\beta+\gamma)}{\gamma}}$$

We again ask the question what the parallel efficiency for the largest problem that can be stored on p nodes is.

$$\begin{aligned} E_p^{\sqrt{p}\times\sqrt{p}}(n_{\max}(p)) &= \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n}\frac{(2\beta+\gamma)}{\gamma}} \\ &= \frac{1}{1 + \frac{\log_2(p)}{2M}\frac{\alpha}{\gamma} + \frac{1}{2\sqrt{M}}\frac{(2\beta+\gamma)}{\gamma}} \end{aligned}$$

so that still

$$\lim_{p\to\infty} E_p^{\sqrt{p}\times\sqrt{p}}(n_{\max}(p)) = \lim_{p\to\infty} \frac{1}{1 + \frac{\log_2(p)}{2M}\frac{\alpha}{\gamma} + \frac{1}{2\sqrt{M}}\frac{(2\beta+\gamma)}{\gamma}} = 0.$$

However, $\log_2 p$ grows very slowly with p and is therefore considered to act much like a constant. In this case $E_p^{\sqrt{p}\times\sqrt{p}}(n_{\max}(p))$ decreases very slowly and the algorithm is considered to be scalable for practical purposes.

Note that when $r = p$ the 2D algorithm becomes the "partitioned by rows" algorithm and when $c = p$ it becomes the "partitioned by columns" algorithm. It is not hard to show that the 2D algorithm is scalable in the sense of the above analysis as long as r/c is kept approximately constant.

Exercise 7.4. Compute the iso-efficiency curve for this operation.

7.3 LU factorization in parallel

The matrix-vector and matrix-product are easy to parallelize in one sense. The elements of the output can all be computed independently and in any order, so we have many degrees of freedom in parallelizing the algorithm. This does not hold for computing an LU factorization, or solving a linear system with the factored matrix.

7.3.1 Solving a triangular system

The solution of a triangular system $y = L^{-1}x$ (with L is lower triangular) is a matrix-vector operation, so it has its $O(N^2)$ complexity in common with the matrix-vector product. However, unlike the product operation, this solution process contains a recurrence relation between the output elements:

$$y_i = \ell_{ii}^{-1}(x_i - \sum_{j<i} \ell_{ij}x_j).$$

This means that parallelization is not trivial. In the case of a sparse matrix special strategies may be possible; see section 7.10. Here we will make a few remarks about general, dense case.

for $k = 1, n-1$:
  for $i = k+1$ to n:
    $a_{ik} \leftarrow a_{ik}/a_{kk}$
  for $i = k+1$ to n:
    for $j = k+1$ to n:
      $a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

Figure 7.5: LU factorization algorithm

Let us assume for simplicity that communication takes no time, and that all arithmetic operations take the same unit time. First we consider the matrix distribution by rows, meaning that processor p stores the elements ℓ_{p*}. With this we can implement the triangular solution as:

- Processor 1 solves $y_1 = \ell_{11}^{-1} x_1$ and sends its value to the next processor.
- In general, processor p gets the values $y_1, \ldots, y_{p-1}$ from processor $p-1$, and computes y_p;
- Each processor p then sends $y_1, \ldots, y_p$ to $p+1$.

Exercise 7.5. Show that this algorithm takes time $2N^2$, just like the sequential algorithm.

This algorithm has each processor passing all computed y_i values to its successor, in a pipeline fashion. However, this means that processor p receives y_1 only at the last moment, whereas that value was computed already in the first step. We can formulate the solution algorithm in such a way that computed elements are made available as soon as possible:

- Processor 1 solve y_1, and sends it to all later processors.
- In general, processor p waits for individual messages with values y_q for $q < p$.
- Processor p then computes y_p and sends it to processors q with $q > p$.

Under the assumption that communication time is negligible, this algorithm can be much faster. For instance, all processors $p > 1$ receive y_1 simultaneously, and can compute $\ell_{p1} y_1$ simultaneously.

Exercise 7.6. Show that this algorithm variant takes a time $O(N)$, if we ignore communication. What is the cost if we incorporate communication in the cost?

Exercise 7.7. Now consider the matrix distribution by columns: processor p stores ℓ_{*p}. Outline the triangular solution algorithm with this distribution, and show that the parallel solve time is $O(N)$.

7.3.2 Factorization, dense case

Analysis of Gaussian elimination, or LU factorization is a little tricky. First of all, Gaussian elimination can be performed in right-looking, left-looking (Doolittle) and Crout variants; see section 5.3.1.1 and [186]. The choice of algorithm, while they all compute the same result, make have computational ramifications.

Let's limit ourselves to the regular right-looking Gaussian elimination of section 5.3.1, reproduced in figure 7.5.

The number of operations in this is

$$\sum_{k=1}^{n} \Big(\sum_{i=k+1}^{n} 1 + \sum_{i=k+1}^{n} \sum_{j=k+1}^{n} 1 \Big) = n^3/3 + O(n^2).$$

The exact calculation does not concern us here; by analogy to integrals you'll see the justice in that factor of $1/3$ on the leading term. Also, we will ignore lower order terms throughout.

If you do a few steps of the algorithm by hand, you see that

$$a_{22} \leftarrow a_{22} - a_{21} * a_{11}^{-1} a_{12}$$
$$\cdots$$
$$a_{33} \leftarrow a_{33} - a_{32} * a_{22}^{-1} a_{23}$$

meaning that there is a chain of dependent operations.

Exercise 7.8. Follow this argument through. Argue that there is a non-trivial *critical path* in the sense of section 2.2.4. What is its length?
In the analysis of the critical path section, what does this critical path imply for the minimum parallel execution time and bounds on speedup?

Now let's take a closer look at how we would actually parallelize this. The major work step is the subblock update:

for $i = k + 1$ to n:
  for $j = k + 1$ to n:
    $a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

In the $k = 1$ step, this involves $(n-1)^2$ operations, and these are all parallel, so you could theoretically use $(n-1)^2$ processing elements (processors, cores, simd lines, etc.)

Exercise 7.9. Continue this reasoning. With $p = n^2$ processing elements each of the (i, j) updates in the subblock can be done simultaneously. To be precise, how long does an arbitrary k iteration take? Summing over all k, what is the resulting T_p, S_p, E_p? How does this relate to the bounds you derived above?
Also, with $p = n$ processing elements you could let each row or column of the subblock update be done in parallel. What is now the time for the kth outer iteration? What is the resulting T_p, S_p, E_p?,

So far we've looked at parallel execution aspects, but without worrying about memory. For a memory analysis we establish the following relations for time and memory in terms of problem size N and processor frequency f:

$$T = \frac{1}{3} N^3 / f, \qquad M = N^2.$$

Note that this formula holds for a single processor computer.

Exercise 7.10. Suppose you buy a processor twice as fast, and you want to do a benchmark run that again takes time T. How much memory do you need?

With p parallel processing elements, the relations become

$$T = \frac{1}{3} N^3 / (pf), \qquad M = N^2.$$

Let's now investigate the implications of an increase in the number of processors, where we assume that each processor has its own memory, containing $M_p = N^2/p$ of the total problem. (Since keep the process itself constant, we can set for simplicity $f = 1$.)

Exercise 7.11. Suppose you have a cluster with p processors, each with M_p memory, can run a Gaussian elimination of an $N \times N$ matrix in time T:

$$T = \frac{1}{3}N^3/p, \qquad M_p = N^2/p.$$

Now you extend the cluster to $2P$ processors, of the same clock speed, and you want to do a benchmark run, again taking time T. How much memory does each node need?
Hint: for the extended cluster:

$$T' = \frac{1}{3}N'^3/p', \qquad M'_p = N'^2/p'.$$

The question becomes to compute M'_p under the given conditions.

7.3.3 Factorization, implementation of the dense case

A full analysis of the scalability of *parallel dense LU factorization* is quite involved, so we will state without further proof that as in the matrix-vector case a two-dimensional distribution is needed.

Exercise 7.12. Analyze the running time, speedup, and efficiency as a function of N, if we assume a one-dimensional distribution, and enough processors to store one column per processor. Show that speedup is limited.
Also perform this analysis for a two-dimensional decomposition where each processor stores one element.

For this reason, an *overdecomposition* is used, where the matrix is divided in more blocks than there are processors, and each processor stores several, non-contiguous, sub-matrices. We illustrate this in figure 7.6

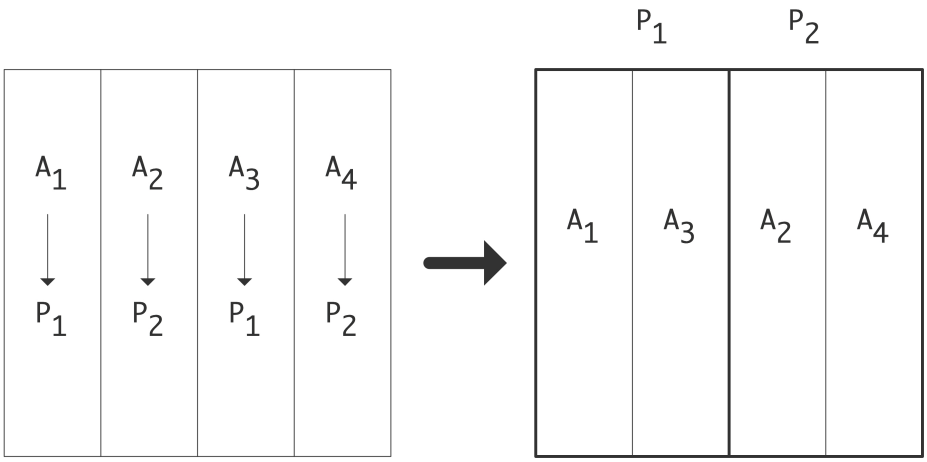


Figure 7.6: One-dimensional cyclic distribution: assignment of four matrix columns to two processors, and the resulting mapping of storage to matrix columns.

where we divide four block columns of a matrix to two processors: each processor stores in a contiguous block of memory two non-contiguous matrix columns.

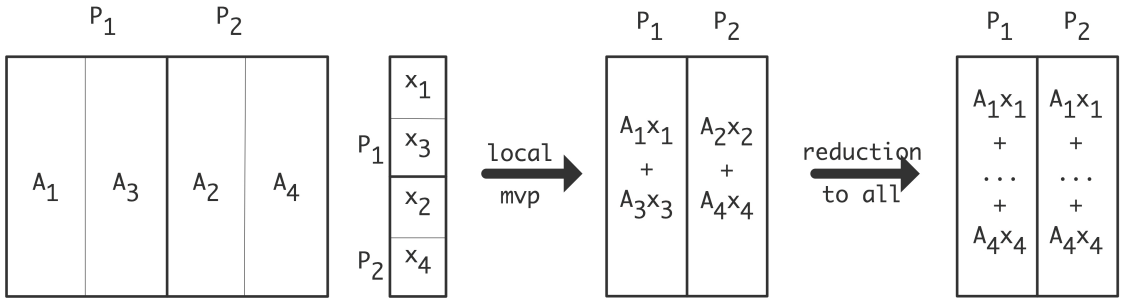


Figure 7.7: Matrix-vector multiplication with a cyclicly distributed matrix.

Next, we illustrate in figure 7.7 that a matrix-vector product with such a matrix can be performed without knowing that the processors store non-contiguous parts of the matrix. All that is needed is that the input vector is also cyclicly distributed.

Exercise 7.13. Now consider a 4×4 matrix and a 2×2 processor grid. Distribute the matrix cyclicly both by rows and columns. Show how the matrix-vector product can again be performed using the contiguous matrix storage, as long as the input is distributed correctly. How is the output distributed? Show that more communication is needed than the reduction of the one-dimensional example.

Specifically, with $P < N$ processors, and assuming for simplicity $N = cP$, we let processor 0 store rows $0, c, 2c, 3c, \ldots$; processor 1 stores rows $1, c+1, 2c+1, \ldots$, et cetera. This scheme can be generalized two a two-dimensional distribution, if $N = c_1 P_1 = c_2 P_2$ and $P = P_1 P_2$. This is called a 2D *cyclic distribution*. This scheme can be further extended by considering block rows and columns (with a small block size), and assigning to processor 0 the *block* rows $0, c, 2c, \ldots$.

Exercise 7.14. Consider a square $n \times n$ matrix, and a square $p \times p$ processor grid, where p divides n without remainder. Consider the overdecomposition outlined above, and make a sketch of matrix element assignment for the specific case $n = 6$, $p = 2$. That is, draw an $n \times n$ table where location (i, j) contains the processor number that stores the corresponding matrix element. Also make a table for each of the processors describing the local to global mapping, that is, giving the global (i, j) coordinates of the elements in the local matrix. (You will find this task facilitated by using zero-based numbering.) Now write functions P, Q, I, J of i, j that describe the global to local mapping, that is, matrix element a_{ij} is stored in location $(I(i,j), J(i,j))$ on processor $(P(i,j), Q(i,j))$.

7.3.4 Factorization, sparse case

Sparse matrix LU factorization in parallel is a notoriously difficult topic. Any type of factorization involves a sequential component, making it non-trivial to begin with. To see the problem with the sparse case in particular, suppose you process the matrix rows sequentially. The dense case has enough elements per row to derive parallelism there, but in the sparse case that number may be very low.

The way out of this problem is to realize that we are not interested in the factorization as such, but rather that we can use it to solve a linear system. Since permuting the matrix gives the same solution, maybe

itself permuted, we can explore permutations that have a higher degree of parallelism.

The topic of matrix orderings already came up in section 5.4.3.5, motivated by fill-in reduction. We will consider orderings with favorable parallelism properties below: nested dissection in section 7.8.1, and multi-color orderings in section 7.8.2.

7.4 Matrix-matrix product

In this section we consider both the sequential and parallel matrix-matrix product, both of which offer some computational challenges.

The *matrix-matrix product* $C \leftarrow A \cdot B$ (or $C \leftarrow A \cdot B + \gamma C$, as it is used in the BLAS) has a simple parallel structure. Assuming all square matrices of size $N \times N$

- the N^3 products $a_{ik}b_{kj}$ can be computed independently, after which
- the N^2 elements c_{ij} are formed by independent sum reductions, each of which take a time $\log_2 N$.

However, considerably more interesting are the questions of data movement:

- With N^3 operations on $O(N^2)$ elements, there is considerable opportunity for *data reuse* (section 1.6.1). We will explore the 'Goto algorithm' in section 7.4.1.
- Depending on the way the matrices are traversed, TLB reuse also needs to be considered (section 1.3.9.2).
- The distributed memory version of the matrix-matrix product is especially tricky. Assuming A, B, C are all distributed over a processor grid, elements of A have to travel through a processor row, and elements of B through a processor columns. We discuss 'Cannon's algorithm' and the outer-product method below.

7.4.1 Goto matrix-matrix product

In section 1.6.1.2 we argued that the *matrix matrix product* (or `dgemm` in *BLAS* terms) has a large amount of possible data reuse: there are $O(n^3)$ operations on $O(n^2)$ data. We will now consider an implementation, due to *Kazushige Goto* [82], that indeed achieves close to peak performance.

The matrix-matrix multiplication algorithm has three loops, each of which we can block (section 6.4.1), giving a six-way nested loop. Since there are no recurrences on the output elements, all resulting loop exchanges are legal. Combine this with the fact that the loop blocking introduces three blocking parameters, and you'll see that the number of potential implementations is enormous. Here we present the global reasoning that underlies the Goto implementation; for a detailed discussion see [82].

We start by writing the product $C \leftarrow A \cdot B$ (or $C \leftarrow C + AB$ according to the BLAS standard) as a sequence of low-rank updates:

$$C_{**} += \sum_k A_{*k}B_{k*}$$

See figure 7.8. Next we derive the 'block-panel' multiplication by multiplying a block of A by a 'sliver' of B; see figure 7.9. Finally, the inner algorithm accumulates a small row $C_{i,*}$, typically of small size such as 4, by accumulating:

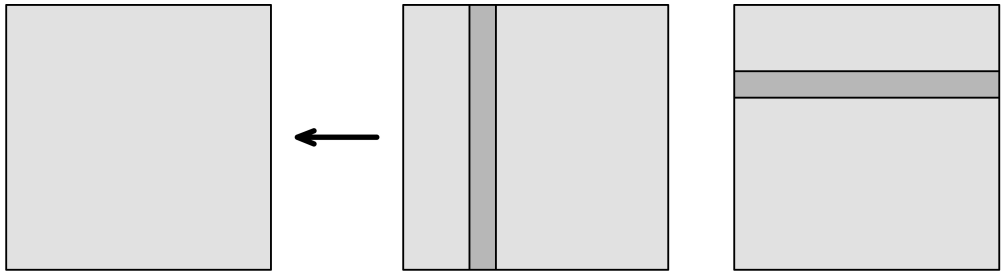


Figure 7.8: Matrix-matrix multiplication as a sequence of low-rank updates.

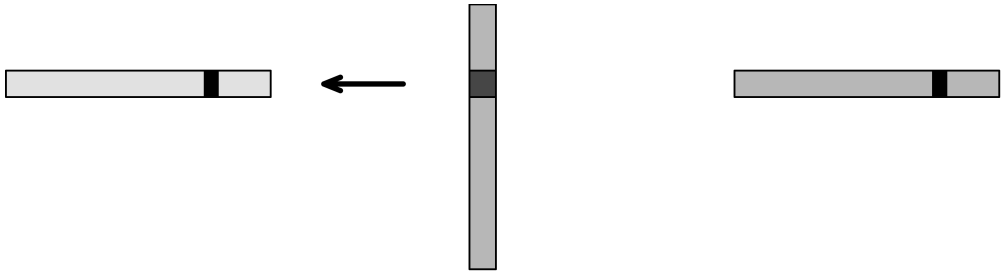


Figure 7.9: The block-panel multiplication in the matrix-matrix algorithm

```
// compute C[i,*] :
for k:
   C[i,*] = A[i,k] * B[k,*]
```

See figure 7.10. Now this algorithm is tuned.

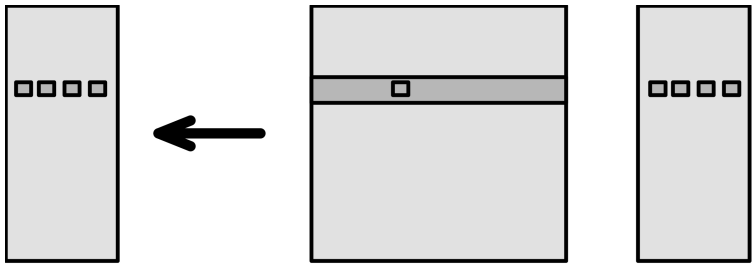


Figure 7.10: The register-resident kernel of the matrix-matrix multiply

- We need enough registers for `C[i,*]`, `A[i,k]` and `B[k,*]`. On current processors that means that we accumulate four or eight elements of C.
- Those elements of C are accumulated, so they stay in register and the only data transfer is loading of the elements of A and B; there are no stores until the elements are fully formed!
- The elements `A[i,k]` and `B[k,*]` stream from L1.
- Since the same block of A is used for many successive slivers of B, we want it to stay resident; we choose the blocksize of A to let it stay in L2 cache. Now we only stream B.

- In order to prevent TLB problems, A is stored by rows. If we start out with a matrix in (Fortran) *column-major* storage, this means we have to make a copy. Since copying is of a lower order of complexity, this cost is amortizable.

This algorithm is available in optimized libraries such as Intel's Math Kernel Library (MKL) [112], BLIS [189], OpenBLas [156]. It can also be implemented in specialized hardware, as in the *Google TPU* [81], giving great energy efficiency.

7.4.2 Cannon's algorithm for the distributed memory matrix-matrix product

In section 7.4.1 we considered the high performance implementation of the single-processor *matrix-matrix product*. We will now briefly consider the distributed memory version of this operation. (It is maybe interesting to note that this is not a generalization based on the matrix-vector product algorithm of section 7.2.)

One algorithm for this operation is known as *Cannon's algorithm*. It assumes a square processor grid where processor (i,j) gradually accumulates the (i,j) block $C_{i,j} = \sum_k A_{i,k}B_{k,j}$; see figure 7.11. (This is intrinsically a *block matrix* operation in the sense of section 5.3.6. Because of the distributed nature of the implementation, here we actually store blocks, rather than considering them only conceptually.)

If you start top-left, you see that processor $(0,0)$ has both A_{00} and B_{00}, so it can immediately start doing a local multiplication. Processor $(0,1)$ has A_{01} and B_{01}, which are not needed together, but if we rotate the second column of B up by one position, processor $(0,1)$ will have A_{01}, B_{11} and those two do need to be multiplied. Similarly, we rotate the third column of B up by two places so that $(0,2)$ contains A_{02}, B_{22}.

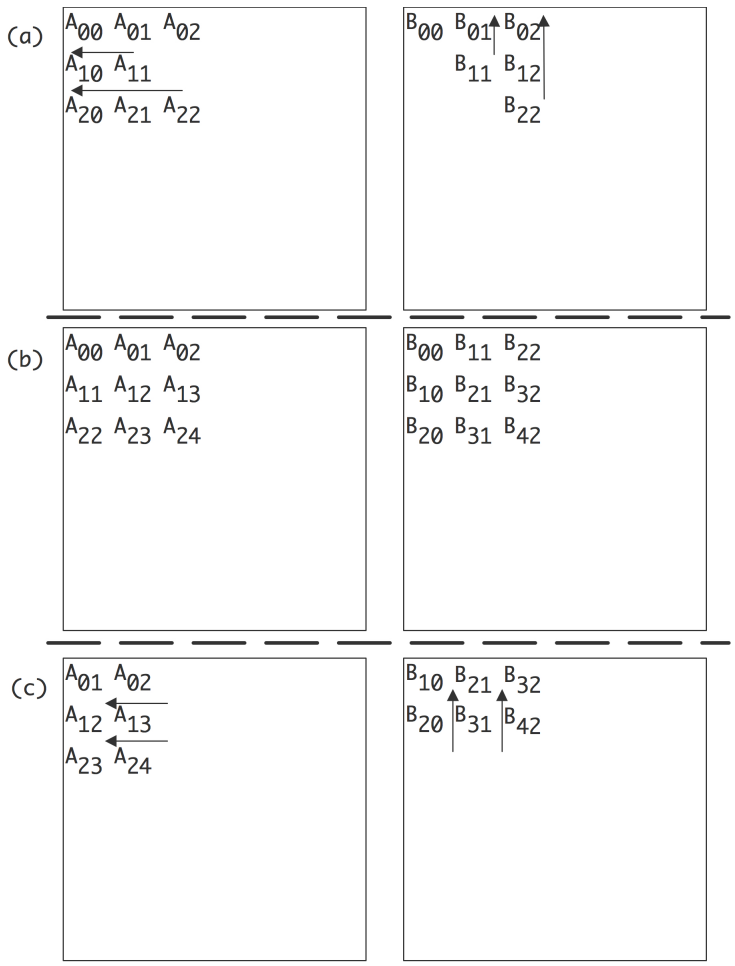


Figure 7.11: Cannon's algorithm for matrix-matrix multiplication: (a) initial rotation of matrix rows and columns, (b) resulting position in which processor (i,j) can start accumulating $\sum_k A_{ik}B_{kj}$, (c) subsequent rotation of A, B for the next term.

How does this story go in the second
row? Processor $(1,0)$ has A_{10}, B_{10} which are not needed together. If we rotate the second row of A one position to the left, it contains A_{11}, B_{10} which are needed for a partial product. And now processor $(1,1)$ has A_{11}, B_{11}.

If we continue this story, we start with a matrix A of which the rows have been rotated left, and B of which the columns have been rotated up. In this setup, processor (i,j) contains $A_{i,i+j}$ and $B_{i+j,j}$, where the addition is done modulo the matrix size. This means that each processor can start by doing a local product.

Now we observe that $C_{ij} = \sum_k A_{ik}B_{kj}$ implies that the next partial product term comes from increasing k by 1. The corresponding elements of A, B can be moved into the processor by rotating the rows and columns by another location.

7.4.3 The outer-product method for the distributed matrix-matrix product

Cannon's algorithm suffered from the requirement of needing a square processors grid. The *outer-product method* is more general. It is based the following arrangement of the matrix-matrix product calculation:

```
for ( k )
  for ( i )
    for ( j )
      c[i,j] += a[i,k] * b[k,j]
```

That is, for each k we take a column of A and a row of B, and computing the rank-1 matrix (or 'outer product') $A_{*,k} \cdot B_{k,*}$. We then sum over k.

Looking at the structure of this algorithm, we notice that in step k, each column j receives $A_{*,k}$, and each row i receives $B_{k,*}$. In other words, elements of $A_{*,k}$ are broadcast through their row, and elements of $B_{k,*}$ are broadcast through their row.

Using the MPI library, these simultaneous broadcasts are realized by having a subcommunicator for each row and each column.

7.5 Sparse matrix-vector product

In linear system solving through iterative methods (see section 5.5) the matrix-vector product is computationally an important kernel, since it is executed in each of potentially hundreds of iterations. In this section we look at performance aspects of the matrix-vector product on a single processor first; the multi-processor case will get our attention in section 7.5.3.

7.5.1 The single-processor sparse matrix-vector product

We are not much worried about the dense matrix-vector product in the context of iterative methods, since one typically does not iterate on dense matrices. In the case that we are dealing with block matrices, refer to section 6.10 for an analysis of the dense product. The sparse product is a lot trickier, since most of that analysis does not apply.

Data reuse in the sparse matrix-vector product There are some similarities between the dense matrix-vector product, executed by rows, and the CRS sparse product (section 5.4.1.4). In both cases all matrix elements are used sequentially, so any cache line loaded is utilized fully. However, the CRS product is worse at least the following ways:

- The indirect addressing requires loading the elements of an integer vector. This implies that the sparse product has more memory traffic for the same number of operations.
- The elements of the source vector are not loaded sequentially, in fact they can be loaded in effectively a random order. This means that a cacheline contains a source element will likely not be fully utilized. Also, the prefetch logic of the memory subsystem (section 1.3.6) cannot assist here.

For these reasons, an application that is computationally dominated by the sparse matrix-vector product can very well be run at $\approx 5\%$ of the peak performance of the processor.

It may be possible to improve this performance if the structure of the matrix is regular in some sense. One such case is where we are dealing with a *block matrix* consisting completely of small dense blocks. This leads at least to a reduction in the amount of indexing information: if the matrix consists of 2×2 blocks we get a $4\times$ reduction in the amount of integer data transferred.

Exercise 7.15. Give two more reasons why this strategy is likely to improve performance.
Hint: cachelines, and reuse.

Such a *matrix tessellation* may give a factor of 2 in performance improvement. Assuming such an improvement, we may adopt this strategy even if the matrix is not a perfect block matrix: if every 2×2 block will contain one zero element we may still get a factor of 1.5 performance improvement [192, 27].

Vectorization in the sparse product In other circumstances bandwidth and reuse are not the dominant concerns:

- On old vector computers, such as old *Cray* machines, memory was fast enough for the processor, but vectorization was paramount. This is a problem for sparse matrices, since the number of zeros in a matrix row, and therefore the vector length, is typically low.
- On GPUs memory bandwidth is fairly high, but it is necessary to find large numbers of identical operations. Matrix can be treated independently, but since the rows are likely of unequal length this is not an appropriate source of parallelism.

For these reasons, a variation on the *diagonal storage* scheme for sparse matrices has seen a revival recently; see section 5.4.1.5. The observation here is that if you sort the matrix rows by the number of rows you get a small number of blocks of rows; each block will be fairly large, and in each block the rows have the same number of elements.

A matrix with such a structure is good for vector architectures [39]. In this case the product is computed by diagonals.

Exercise 7.16. Write pseudo-code for this case. How did the sorting of the rows improve the situation?

This sorted storage scheme also solves the problem we noted on GPUs [20]. In this case we the traditional CRS product algorithm, and we have an amount of parallelism equal to the number of rows in a block.

Of course there is the complication that we have permuted the matrix: the input and output vectors will need to be permuted accordingly. If the product operation is part of an iterative method, doing this permutation back and forth in each iteration will probably negate any performance gain. Instead we could permute the whole linear system and iterate on the permuted system.

Exercise 7.17. Can you think of reasons why this would work? Reasons why it wouldn't?

7.5.2 The parallel sparse matrix-vector product

In section 5.4 you saw a first discussion of sparse matrices, limited to use on a single processor. We will now go into parallelism aspects.

The dense matrix-vector product, as you saw above, required each processor to communicate with every other, and to have a local buffer of essentially the size of the global vector. In the sparse case, considerably less buffer space, as well as less communication, is needed.

Let us analyze the distributed memory parallel sparse matrix-vector product in some detail. We will assume that the matrix is distributed by block rows, where processor p owns (in the sense of computing the output there) the matrix rows with indices in some set I_p.

The line $y_i = y_i + a_{ij}x_j$, which in the dense case is executed for all j, and therefore induces an allgather of the right-hand side, now has to take into account that a_{ij} can be zero, so for some pairs $i \in I_p, j \notin I_p$ no communication will be needed. If both $i, j \in I_p$, we have a traditional single-processor product; the interesting case is $i \in I_p, j \notin I_p$ where x_j needs to be communicated to processor p from another processor.

Declaring for each $i \in I_p$ a sparsity pattern set

$$S_{p;i} = \{j : \ j \notin I_p, a_{ij} \neq 0\}$$

our multiplication instruction becomes

$$\begin{cases} y_i += a_{ij} \,\mathrm{local}(x_j) & \text{for } j \in I_p, a_{ij} \neq 0 \\ y_i += a_{ij} \,\mathrm{communicated}(x_j) & \text{for } j \in S_{p;i}. \end{cases}$$

If we want to avoid a flood of small messages, we combine all communication into a single message per processor. Defining

$$S_p = \cup_{i \in I_p} S_{p;i},$$

the algorithm now becomes:

- Collect all necessary off-processor elements x_j with $j \in S_p$ into one buffer;
- Perform the matrix-vector product, reading all elements of x from local storage.

Recall figure 4.1, which illustrates a discretized boundary value problem on the simplest domain, a square, and let us now parallelize it. We do this by partitioning the domain; each processor gets the matrix rows corresponding to its subdomain. Figure 7.12 shows how this gives rise to connections between processors: the elements a_{ij} with $i \in I_p, j \notin I_p$ are now the 'legs' of the stencil that reach beyond a processor boundary. The set S_p of all such j, defined above, is known as the *halo region* of a processor; see figure 7.13.

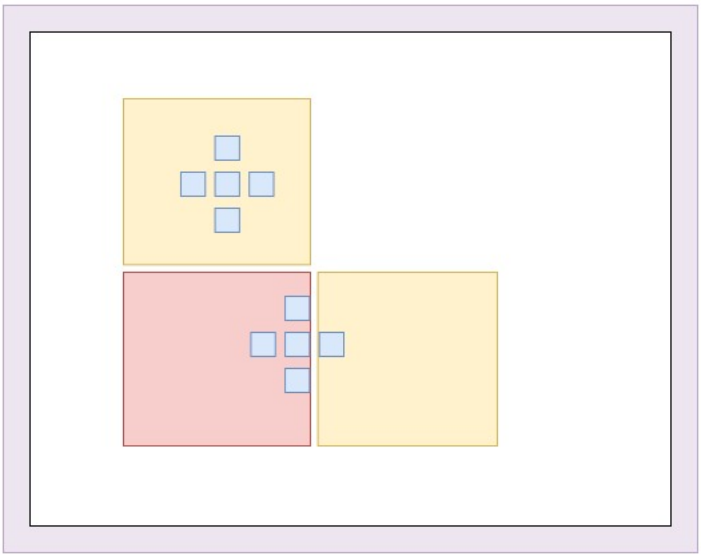


Figure 7.12: A difference stencil applied to a two-dimensional square domain, distributed over processors. A cross-processor connection is indicated..

Exercise 7.18. Show that a one-dimensional partitioning of the domain leads to a partitioning of the matrix into block rows with a contiguous block for each process. Show that a two-dimensional partitioning of the domain does not give this structure. You can do this in the abstract, or you can illustrate it: take a 4×4 domain (giving a matrix of size 16), and partition it over 4 processors. The one-dimensional domain partitioning corresponds to giving each processor one line out of the domain, while the two-dimensional partitioning gives each processor a 2×2 subdomain. Draw the matrices for these two cases.

Exercise 7.19. Figure 7.14 depicts a sparse matrix of size N with a halfbandwidth $n = \sqrt{N}$. That is,

$$|i - j| > n \Rightarrow a_{ij} = 0.$$

We make a one-dimensional distribution of this matrix over p processors, where $p = n = \sqrt{N}$.
Show that the matrix-vector product using this scheme is weakly scalable by computing the efficiency as function of the number of processors

$$E_p = \frac{T_1}{pT_p}.$$

Why is this scheme not really weak scaling, as it is commonly defined?

One crucial observation about the parallel sparse matrix-vector product is that, for each processor, the number of other processors it is involved with is strictly limited. This has implications for the efficiency of the operation.

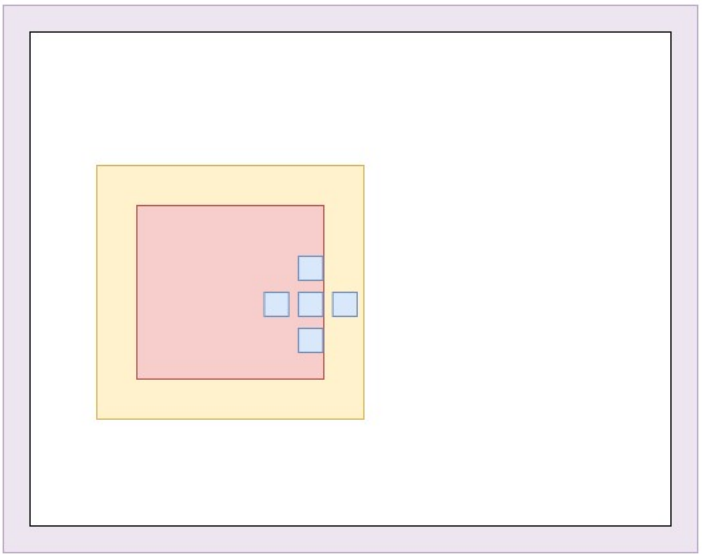


Figure 7.13: The halo region of a processor, induced by a stencil.

In the case of the dense matrix-vector product (section 7.2.3), partitioning the matrix over the processors by (block) rows does not lead to a scalable algorithm. In global terms we can explain this by the fact that, as the number of processors increases, the messsage size does not go down similarly to the work per processor.

Exercise 7.20. Take a square domain and a partitioning of the variables of the processors as in figure 7.12. What is the maximum number of neighbors a processor needs to communication with for the box stencil in figure 4.3? In three space dimensions, what is the number of neighbors if a 7-point central difference stencil is used?

The observation that each processor communicates with just a few neighbors stays intact if we go beyond square domains to more complicated physical objects. If a processor receives a contiguous, more or less convex subdomain, the number of its neighbors will be limited. This implies that even in complicated problems each processor will only communicate with a small number of other processors. Compare this to the dense case where each processor had to receive data from *every* other processor. It is obvious that the sparse case is far more friendly to the interconnection network. (The fact that it also more common for large systems may influence the choice of network to install if you are about to buy a new parallel computer.)

7.5.3 Parallel efficiency of the sparse matrix-vector product

For square domains we can make the above argument formal. Let the unit domain $[0,1]^2$ be partitioned over P processors in a $\sqrt{P} \times \sqrt{P}$ grid. From figure 7.12 we see that every processor communicates with at most four neighbors. Let the amount of work per processor be w and the communication time with each neighbor c. Then the time to perform the total work on a single processor is $T_1 = Pw$, and the parallel

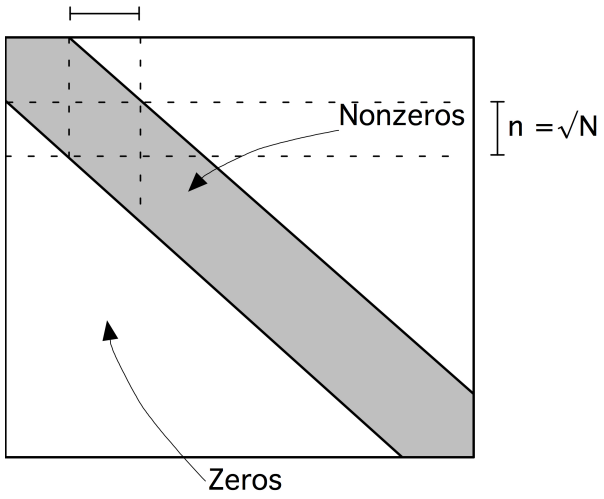


Figure 7.14: Band matrix with halfbandwidth $\sqrt{N}$.

time is $T_P = w + 4c$, giving a speed up of

$$S_P = Pw/(w + 4c) = P/(1 + 4c/w) \approx P(1 - 4c/w).$$

Exercise 7.21. Express c and w as functions of N and P, and show that the speedup is asymptotically optimal, under weak scaling of the problem.

Exercise 7.22. In this exercise you will analyze the parallel sparse matrix-vector product for a hypothetical, but realistic, parallel machine. Let the machine parameters be characterized by (see section 1.3.2):

- Network *Latency:* $\alpha = 1\mu s = 10^{-6}s$.
- Network *Bandwidth:* $1Gb/s$ corresponds to $\beta = 10^{-9}$.
- *Computation rate:* A per-core flops rate of $1G$flops means $\gamma = 10^{-9}$. This number may seem low, but note that the matrix-vector product has less reuse than the matrix-matrix product, which can achieve close to peak performance, and that the sparse matrix-vector product is even more bandwidth-bound.

We perform a combination of asymptotic analysis and deriving specific numbers. We assume a cluster of 10^4 single-core processors[3]. We apply this to a five-point stencil matrix of size $N = 25 \cdot 10^{10}$. This means each processor stores $5 \cdot 8 \cdot N/p = 10^9$ bytes. If the matrix comes from a problem on a square domain, this means the domain was size $n \times n$ where $n = \sqrt{N} = 5 \cdot 10^5$.

Case 1. Rather than dividing the matrix, we divide the domain, and we do this first by horizontal slabs of size $n \times (n/p)$. Argue that the communication complexity is $2(\alpha + n\beta)$ and computation complexity is $10 \cdot n \cdot (n/p)$. Show that the resulting computation outweighs the communication by a factor 250.

3. Introducing multicore processors would complicate the story, but since the number of cores is $O(1)$, and the only way to grow a cluster is by adding more networked nodes, it does not change the asymptotic analysis.

Case 2. We divide the domain into patches of size $(n/\sqrt{p}) \times (n/\sqrt{p})$. The memory and computation time are the same as before. Derive the communication time and show that it is better by a factor of 50.
Argue that the first case does not weakly scale: under the assumption that N/p is constant the efficiency will go down. (Show that speedup still goes up asymptotically as $\sqrt{p}$.) Argue that the second case does scale weakly.

The argument that a processor will only connect with a few neighbors is based on the nature of the scientific computations. It is true for FDM and FEM methods. In the case of the *Boundary Element Method (BEM)*, any subdomain needs to communicate with everything in a radius r around it. As the number of processors goes up, the number of neighbors per processor will also go up.

Exercise 7.23. Give a formal analysis of the speedup and efficiency of the BEM algorithm. Assume again a unit amount of work w per processor and a time of communication c per neighbor. Since the notion of neighbor is now based on physical distance, not on graph properties, the number of neighbors will go up. Give T_1, T_p, S_p, E_p for this case.

There are also cases where a sparse matrix needs to be handled similarly to a dense matrix. For instance, *Google*'s *PageRank* algorithm (see section 10.4) has at its heart the repeated operation $x \leftarrow Ax$ where A is a sparse matrix with $A_{ij} \neq 0$ if web page j links to page i; see section 10.4. This makes A a very sparse matrix, with no obvious structure, so every processor will most likely communicate with almost every other.

7.5.4 Memory behavior of the sparse matrix-vector product

In section 6.10 you saw an analysis of the sparse matrix-vector product in the dense case, on a single processor. Some of the analysis carries over immediately to the sparse case, such as the fact that each matrix element is used only once and that the performance is bound by the bandwidth between processor and memory.

Regarding reuse of the input and the output vector, if the matrix is stored by rows, such as in *CRS* format (section 5.4.1.3), access to the output vector will be limited to one write per matrix row. On the other hand, the loop unrolling trick for getting reuse of the input vector can not be applied here. Code that combines two iterations is as follows:

```
for (i=0; i<M; i+=2) {
  s1 = s2 = 0;
  for (j) {
    s1 = s1 + a[i][j] * x[j];
    s2 = s2 + a[i+1][j] * x[j];
  }
  y[i] = s1; y[i+1] = s2;
}
```

The problem here is that if a_{ij} is nonzero, it is not guaranteed that $a_{i+1,j}$ is nonzero. The irregularity of the sparsity pattern makes optimizing the matrix-vector product hard. Modest improvements are possible by identifying parts of the matrix that are small dense blocks [27, 44, 191].

On a *GPU* the sparse matrix-vector product is also limited by memory bandwidth. Programming is now harder because the GPU has to work in data parallel mode, with many active threads.

An interesting optimization becomes possible if we consider the context in which the sparse matrix-vector product typically appears. The most common use of this operation is in iterative solution methods for linear systems (section 5.5), where it is applied with the same matrix in possibly hundreds of iterations. Thus we could consider leaving the matrix stored on the GPU and only copying the input and output vectors for each product operation.

7.5.5 The transpose product

In section 5.4.1.3 you saw that the code for both the regular and the transpose matrix-vector product are limited to loop orderings where rows of the matrix are traversed. (In section 1.6.2 you saw a discussion of computational effects of changes in loop order; in this case we are limited to row traversal by the storage format.)

In this section we will briefly look at the parallel transpose product. Equivalently to partitioning the matrix by rows and performing the transpose product, we look at a matrix stored and partitioned by columns and perform the regular product.

The algorithm for the product by columns can be given as:

$y \leftarrow 0$
for j:
    for i:
        $y_i \leftarrow y_i + a_{ij}x_j$

Both in shared and distributed memory we distribute outer iterations over processors. The problem is then that each outer iteration updates the whole output vector. This is a problem: with shared memory it leads to multiple writes to locations in the output and in distributed memory it requires communication that is as yet unclear.

One way to solve this would be to allocate a private output vector $y^{(p)}$ for each process:

$y^{(p)} \leftarrow 0$
for $j \in$ the rows of processor p
    for all i:
        $y_i^{(p)} \leftarrow y_i^{(p)} + a_{ji}x_j$

after which we sum $y \leftarrow \sum_p y^{(p)}$.

7.5.6 Setup of the sparse matrix-vector product

Above we defined sets of processes S_p that contain data to be sent to process p. It is easy enough for process p to construct this set from its part of the sparse matrix.

Exercise 7.24. Assume that the matrix is divided over the processors by block rows; see figure 7.2 for an illustration. Also assume that each processor knows which rows each other processor stores. (How would you implement that knowledge?)

Sketch the algorithm by which a processor can find out who it will be receiving from; this algorithm should not involve any communication itself.

However, there is an asymmetry between sending and receiving. While it is fairly easy for a processor to find out what other processors it will be receiving from, discovering who to send *to* is harder.

Exercise 7.25. Argue that this is easy in the case of a *structurally symmetric matrix*: $a_{ij} \neq 0 \Leftrightarrow a_{ji} \neq 0$.

In the general case, a processor can in principle be asked to send to any other, so the simple algorithm is as follows:

- Each processor makes an inventory of what non-local indices it needs. Under the above assumption that it knows what range of indices each other processor owns, it then decides which indices to get from what neighbors.
- Each processor sends a list of indices to each of its neighbors; this list will be empty for most of the neighbors, but we can not omit sending it.
- Each processor then receives these lists from all others, and draws up lists of which indices to send.

You will note that, even though the communication during the matrix-vector product involves only a few neighbors for each processor, giving a cost that is $O(1)$ in the number of processors, the setup involves all-to-all communications, which have time complexity $O(\alpha P)$

If a processor has only a few neighbors, the above algorithm is wasteful. Ideally, you would want space and running time proportional to the number of neighbors. We could bring the receive time in the setup if we knew how many messages were to be expected. That number can be found:

- Each processor makes an array `need` of length P, where `need[i]` is 1 if the processor needs any data from processor i, and zero otherwise.
- A *reduce-scatter* collective on this array, with a sum operator, then leaves on each processor a number indicating how many processors need data from it.
- The processor can execute that many receive calls.

The reduce-scatter call has time complexity $\alpha \log P + \beta P$, which is of the same order as the previous algorithm, but probably with a lower proportionality constant.

The time and space needed for the setup can be reduced to $O(\log P)$ with some sophisticated trickery [61, 107].

7.6 Computational aspects of iterative methods

All iterative methods feature the following operations:

- A matrix-vector product; this was discussed for the sequential case in section 5.4 and for the parallel case in section 7.5. In the parallel case, construction of FEM matrices has a complication that we will discuss in section 7.6.2.
- The construction of the preconditioner matrix $K \approx A$, and the solution of systems $Kx = y$. This was discussed in the sequential case in section 5.5.6. We will go into parallelism aspects in section 7.7.
- Some vector operations (including inner products, in general). These will be discussed next.

7.6.1 Vector operations

There are two types of vector operations in a typical iterative method: vector additions and inner products.

Exercise 7.26. Consider the CG method of section 5.5.11, figure 5.11, applied to the matrix from a 2D BVP; equation (4.56), First consider the unpreconditioned case $M = I$. Show that there is a roughly equal number of floating point operations are performed in the matrix-vector product and in the vector operations. Express everything in the matrix size N and ignore lower order terms. How would this balance be if the matrix had 20 nonzeros per row?
Next, investigate this balance between vector and matrix operations for the FOM scheme in section 5.5.9. Since the number of vector operations depends on the iteration, consider the first 50 iterations and count how many floating point operations are done in the vector updates and inner product versus the matrix-vector product. How many nonzeros does the matrix need to have for these quantities to be equal?

Exercise 7.27. Flop counting is not the whole truth. What can you say about the efficiency of the vector and matrix operations in an iterative method, executed on a single processor?

7.6.1.1 Vector additions

The vector additions are typically of the form $x \leftarrow x + \alpha y$ or $x \leftarrow \alpha x + y$. If we assume that all vectors are distributed the same way, this operation is fully parallel.

7.6.1.2 Inner products

Inner products are vector operations, but they are computationally more interesting than updates, since they involve communication.

When we compute an inner product, most likely every processor needs to receive the computed value. We use the following algorithm:

for *processor* p **do**
  compute $a_p \leftarrow x_p^t y_p$ where x_p, y_p are the part of x, y stored on processor p
do a global all-reduction to compute $a = \sum_p a_p$
broadcast the result

Algorithm 1: Compute $a \leftarrow x^t y$ where x, y are distributed vectors.

The `Allreduce`) combines data over all processors, so they have a communication time that increases with the number of processors. This makes the inner product potentially an expensive operation (also they form a barrier-like synchronization), and people have suggested a number of ways to reducing their impact on the performance of iterative methods.

Exercise 7.28. Iterative methods are typically used for sparse matrices. In that context, you can argue that the communication involved in an inner product can have a larger influence on overall performance than the communication in the matrix-vector product. What is the complexity of the matrix-vector product and the inner product as a function of the number of processors?

Here are some of the approaches that have been taken.

- The CG method has two inner products per iteration that are inter-dependent. It is possible to rewrite the method so that it computes the same iterates (in exact arithmetic, at least) but so that the two inner products per iteration can be combined. See [33, 40, 150, 199].
- It may be possible to overlap the inner product calculation with other, parallel, calculations [42].
- In the GMRES method, use of the classical Gram-Schmidt (GS) method takes far fewer independent inner product than the modified GS method, but it is less stable. People have investigated strategies for deciding when it is allowed to use the classic GS method [133].

Since computer arithmetic is not associative, inner products are a prime source of results that differ when the same calculation is executed of two different processor configurations. In section 3.6.5 we sketched a solution.

7.6.2 Finite element matrix construction

The *FEM* leads to an interesting issue in parallel computing. For this we need to sketch the basic outline of how this method works. The FEM derives its name from the fact that the physical objects modeled are divided into small two or three dimensional shapes, the elements, such as triangles and squares in 2D, or pyramids and bricks in 3D. On each of these, the function we are modeling is then assumed to polynomial, often of a low degree, such as linear or bilinear.

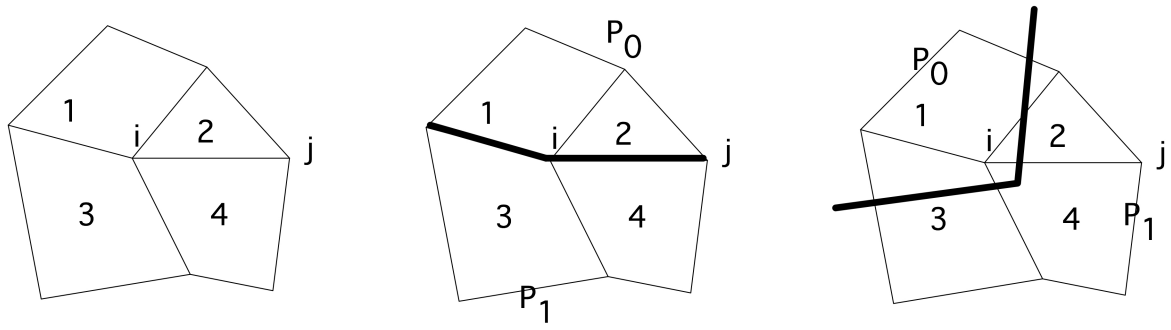


Figure 7.15: A finite element domain, parallelization of the matrix construction, and parallelization of matrix element storage.

The crucial fact is that a matrix element a_{ij} is then the sum of computations, specifically certain integrals, over all elements that contain both variables i and j:

$$a_{ij} = \sum_{e\,:\,i,j\in e} a_{ij}^{(e)}.$$

The computations in each element share many common parts, so it is natural to assign each element e uniquely to a processor P_e, which then computes all contributions $a_{ij}^{(e)}$. In figure 7.15 element 2 is assigned to processor 0 and element 4 to processor 1.

Now consider variables i and j and the matrix element a_{ij}. It is constructed as the sum of computations over domain elements 2 and 4, which have been assigned to different processors. Therefore, no matter what processor row i is assigned to, at least one processor will have to communicate its contribution to matrix element a_{ij}.

Clearly it is not possibly to make assignments P_e of elements and P_i of variables such that P_e computes in full the coefficients a_{ij} for all $i \in e$. In other words, if we compute the contributions locally, there needs to be some amount of communication to assemble certain matrix elements. For this reason, modern linear algebra libraries such as PETSc *Parallel Programming book, section 31.4.3* allow any processor to set any matrix element.

7.6.3 A simple model for iterative method performance

Above, we have already remarked that iterative methods have very little opportunity for data reuse, and they are therefore characterized as *bandwidth-bound* algorithms. This allows us to make a simple prediction as to the *floating point performance* of iterative methods. Since the number of iterations of an iterative method is hard to predict, by performance here we mean the performance of a single iteration. Unlike with *direct methods for linear systems* (see for instance section 7.8.1), the number of flops to solution is very hard to establish in advance.

First we argue that we can restrict ourselves to the *performance of the sparse matrix vector product*: the time spent in this operation is considerably more than in the vector operations. Additionally, most preconditioners have a computational structure that is quite similar to the matrix-vector product.

Let us then consider the *performance of the CRS matrix-vector product*.

- First we observe that the arrays of matrix elements and column indices have no reuse, so the performance of loading them is completely determined by the available bandwidth. Caches and prefetch streams only hide the latency, but do not improve the bandwidth. For each multiplication of matrix element times input vector element we load one floating point number and one integer. Depending one whether the indices are 32-bit or 64-bit, this means 12 or 16 bytes loaded for each multiplication.
- The demands for storing the result vector are less important: an output vector element is written only once for each matrix row.
- The input vector can also be ignored in the bandwidth calculation. At first sight you might think that indirect indexing into the input vector is more or less random and therefore expensive. However, let's take the operator view of a matrix-vector product and consider the space domain of the PDE from which the matrix stems; see section 4.2.3. We now see that the seemingly random indexing is in fact into vector elements that are grouped closely together. This means that these vector elements will likely reside in L3 cache, and therefore accessible with a higher bandwidth (say, by a factor of at least 5) than data from main memory.

For the parallel performance of the sparse matrix-vector product we consider that in a PDE context each processor communicates only with a few neighbors. Furthermore, a surface-to-volume argument shows that the message volume is of a lower order than the on-node computation.

In sum, we conclude that a very simple model for the sparse matrix vector product, and thereby for the

whole iterative solver, consists of measuring the effective bandwidth and computing the performance as one addition and one multiplication operation per 12 or 16 bytes loaded.

7.7 Parallel preconditioners

Above (section 5.5.6 and in particular 5.5.6.1) we saw a couple of different choices of K. In this section we will begin the discussion of parallelization strategies. The discussion is continued in detail in the next sections.

7.7.1 Jacobi preconditioning

The Jacobi method (section 5.5.3) uses the diagonal of A as preconditioner. Applying this is as parallel as is possible: the statement $y \leftarrow K^{-1}x$ scales every element of the input vector independently. Unfortunately the improvement in the number of iterations with a Jacobi preconditioner is rather limited. Therefore we need to consider more sophisticated methods such ILU. Unlike with the Jacobi preconditioner, parallelism is then not trivial.

7.7.2 Gauss-Seidel and SOR

Regarding GS and SOR we can remark that they are not often used as preconditioners since they are intrinsically unsymmetric. For SOR moreover there is the problem of determining a suitable value for ω. The symmetrized Symmetric Successive Over-Relaxation (SSOR) (is similar enough to ILU) that we will not discuss it separately.

These methods do find a use as *multigrid smoother*s, and here an interesting observation is to be made. Consider the solution of a GS system of equations by wavefronts (section 7.10.1), and imagine several iterations to be active simultaneously. It is not hard to see that this is equivalent to a multi-color scheme, be it with a different starting vector. See [1] for the full argument.

7.7.3 The trouble with ILU in parallel

Above we saw that, in a flop counting sense, applying an ILU preconditioner (section 5.5.6.1) is about as expensive as doing a matrix-vector product. This is no longer true if we run our iterative methods on a parallel computer.

At first glance the operations are similar. A matrix-vector product $y = Ax$ looks like

```
for i=1..n
  y[i] = sum over j=1..n a[i,j]*x[j]
```

In parallel this would look like

```
for i=myfirstrow..mylastrow
  y[i] = sum over j=1..n a[i,j]*x[j]
```

Suppose that a processor has local copies of all the elements of A and x that it will need, then this operation is fully parallel: each processor can immediately start working, and if the work load is roughly equal, they will all finish at the same time. The total time for the matrix-vector product is then divided by the number of processors, making the speedup more or less perfect.

Consider now the forward solve $Lx = y$, for instance in the context of an ILU preconditioner:

```
for i=1..n
  x[i] = (y[i] - sum over j=1..i-1 ell[i,j]*x[j]) / a[i,i]
```

We can simply write the parallel code:

```
for i=myfirstrow..mylastrow
  x[i] = (y[i] - sum over j=1..i-1 ell[i,j]*x[j]) / a[i,i]
```

but now there is a problem. We can no longer say 'suppose a processor has local copies of everything in the right hand side', since the vector x appears both in the left and right hand side. While the matrix-vector product is in principle fully parallel over the matrix rows, this triangular solve code is recursive, hence sequential.

In a parallel computing context this means that, for the second processor to start, it needs to wait for certain components of x that the first processor computes. Apparently, the second processor can not start until the first one is finished, the third processor has to wait for the second, and so on. The disappointing conclusion is that in parallel only one processor will be active at any time, and the total time is the same as for the sequential algorithm. This is actually not a big problem in the dense matrix case, since parallelism can be found in the operations for handling a single row (see section 7.12), but in the sparse case it means we can not use incomplete factorizations without some redesign.

In the next few subsections we will see different strategies for finding preconditioners that perform efficiently in parallel.

7.7.4 Block Jacobi methods

Various approaches have been suggested to remedy this sequentiality the triangular solve. For instance, we could simply let the processors ignore the components of x that should come from other processors:

```
for i=myfirstrow..mylastrow
  x[i] = (y[i] - sum over j=myfirstrow..i-1 ell[i,j]*x[j])
         / a[i,i]
```

This is not mathematically equivalent to the sequential algorithm (technically, it is called a *block Jacobi* method with ILU as the *local solve*), but since we're only looking for an approximation $K \approx A$, this is simply a slightly cruder approximation.

Exercise 7.29. Take the Gauss-Seidel code you wrote above, and simulate a parallel run. What is the effect of increasing the (simulated) number of processors?

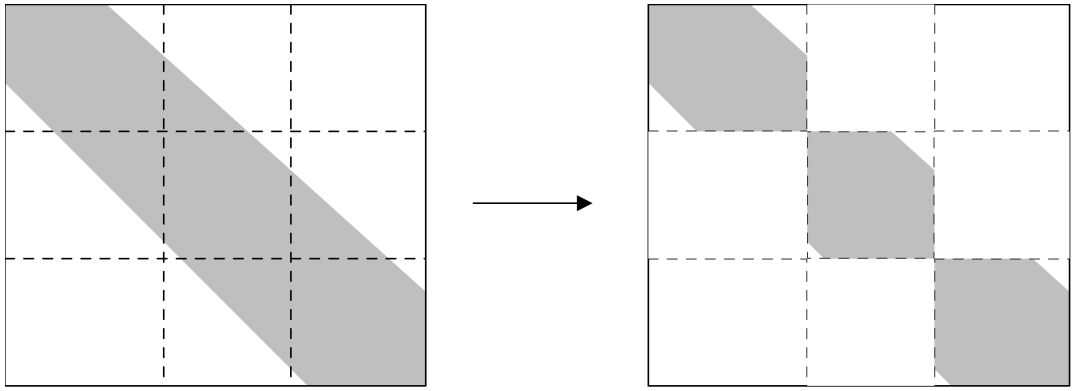


Figure 7.16: Sparsity pattern corresponding to a block Jacobi preconditioner.

The idea behind block methods can be appreciated pictorially; see figure 7.16. In effect, we make an ILU of the matrix that we get by ignoring all connections between processors. Since in a BVP all points influence each other (see section 4.2.1), using a less connected preconditioner will increase the number of iterations if executed on a sequential computer. However, block methods are parallel and, as we observed above, a sequential preconditioner is very inefficient in a parallel context, so we put up with this increase in iterations.

7.7.5 Parallel ILU

The Block Jacobi preconditioner operates by decoupling domain parts. While this may give a method that is highly parallel, it may give a higher number of iterations than a true ILU preconditioner. Fortunately it is possible to have a parallel ILU method. Since we need the upcoming material on re-ordering of variables, we postpone this discussion until section 7.8.2.3.

7.8 Ordering strategies and parallelism

In the foregoing we have remarked on the fact that solving a linear system of equations is inherently a recursive activity. For dense systems, the number of operations is large enough compared to the recursion length that finding parallelism is fairly straightforward. Sparse systems, on the other hand, take more sophistication. In this section we will look at a number of strategies for reordering the equations (or, equivalently, permuting the matrix) that will increase the available parallelism.

These strategies can all be considered as variants of Gaussian elimination. By making incomplete variants of them (see section 5.5.6.1), all these strategies also apply to constructing preconditioners for iterative solution methods.

7.8.1 Nested dissection

In the previous discussions we typically looked at square domains and a *lexicographic ordering* of the variables. In this section you will see the *nested dissection ordering* of variables, which was initially designed

as a way to reduce fill-in; see [74]. More importantly to us, it is also advantageous in a parallel computing context, as we will now see.

Nested dissection is a recursive process for determining a nontrivial ordering of the unknowns in a domain. In the first step, the computational domain is split in two parts, with a dividing strip between them; see figure 7.17. To be precise, the *separator* is wide enough that there are no connections between the left

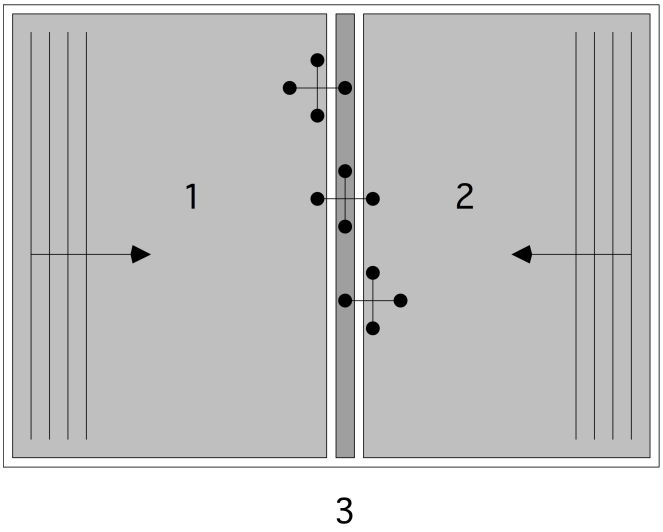


Figure 7.17: Domain dissection into two unconnected subdomains and a separator.

and right *subdomain*. The resulting matrix A^{DD} has a 3×3 structure, corresponding to the three parts of the domain. Since the subdomains Ω_1 and Ω_2 are not connected, the submatrices A^{DD}_{12} and A^{DD}_{21} are zero.

$$A^{\mathrm{DD}} = \begin{pmatrix} A_{11} & \emptyset & A_{13} \\ \emptyset & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \left(\begin{array}{ccccc|ccccc|c} \star & \star & & & & & & & & & 0 \\ \star & \star & \star & & & & & & & & \vdots \\ & \ddots & \ddots & \ddots & & & & \emptyset & & & \vdots \\ & & \star & \star & \star & & & & & & 0 \\ & & & \star & \star & & & & & & \star \\ \hline & & & & & \star & \star & & & & 0 \\ & & & & & \star & \star & \star & & & \vdots \\ & & \emptyset & & & & \ddots & \ddots & \ddots & & \vdots \\ & & & & & & & \star & \star & \star & 0 \\ & & & & & & & & \star & \star & \star \\ \hline 0 & \cdots & \cdots & 0 & \star & 0 & \cdots & \cdots & 0 & \star & \star \end{array} \right) \begin{array}{l} \left. \vphantom{\begin{array}{c} a \\ a \\ a \\ a \\ a \end{array}} \right\} (n^2-n)/2 \\ \left. \vphantom{\begin{array}{c} a \\ a \\ a \\ a \\ a \end{array}} \right\} (n^2-n)/2 \\ \left. \vphantom{a} \right\} n \end{array}$$

This process of dividing the domain by a separator is also called *domain decomposition* or *substructuring*, although this name is also associated with the mathematical analysis of the resulting matrices [14]. In this example of a rectangular domain it is of course trivial to find a separator. However, for the type of equations we get from BVPs it is usually feasible to find a separator efficiently for any domain [141]; see also section 20.6.2.

Let us now consider the LU factorization of this matrix. If we factor it in terms of the 3×3 block structure, we get

$$A^{\mathrm{DD}} = LU = \begin{pmatrix} I & & \\ \emptyset & I & \\ A_{31}A_{11}^{-1} & A_{32}A_{22}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{11} & \emptyset & A_{13} \\ & A_{22} & A_{23} \\ & & S_{33} \end{pmatrix}$$

where

$$S_{33} = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23}.$$

The important fact here is that

- the contributions $A_{31}A_{11}^{-1}A_{13}$ and $A_{32}A_{22}^{-1}A_{23}$ can be computed simultaneously, so the factorization is largely parallel; and
- both in the forward and backward solve, components 1 and 2 of the solution can be computed simultaneously, so the solution process is also largely parallel.

The third block can not trivially be handled in parallel, so this introduces a sequential component in the algorithm. Let's take a closer look at the structure of S_{33}.

Exercise 7.30. In section 5.4.3.1 you saw the connection between LU factorization and graph theory: eliminating a node leads to a graph with that node removed, but with certain new connections added. Show that, after eliminating the first two sets Ω_1, Ω_2 of variables, the graph of the remaining matrix on the separator will be fully connected.

The upshot is that after eliminating all the variables in blocks 1 and 2 we are left with a matrix S_{33} that is fully dense of size $n \times n$.

The introduction of a separator gave us a factorization that was two-way parallel. Now we iterate this process: we put a separator inside blocks 1 and 2 (see figure 7.18), which gives the following matrix structure:

$$A^{\mathrm{DD}} = \left(\begin{array}{cccc|cc|c} A_{11} & & & & A_{15} & & A_{17} \\ & A_{22} & & & A_{25} & & A_{27} \\ & & A_{33} & & & A_{36} & A_{37} \\ & & & A_{44} & & A_{46} & A_{47} \\ \hline A_{51} & A_{52} & & & A_{55} & & A_{57} \\ & & A_{63} & A_{64} & & A_{66} & A_{67} \\ \hline A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{array}\right)$$

(Note the similarities with the 'arrow' matrix in section 5.4.3.4, and recall the argument that this led to lower fill-in.) The LU factorization of this is:

$$\left(\begin{array}{cccc|cc|c} I & & & & & & \\ & I & & & & & \\ & & I & & & & \\ & & & I & & & \\ \hline A_{51}A_{11}^{-1} & A_{52}A_{22}^{-1} & & & I & & \\ & & A_{63}A_{33}^{-1} & A_{64}A_{44}^{-1} & & I & \\ \hline A_{71}A_{11}^{-1} & A_{72}A_{22}^{-1} & A_{73}A_{33}^{-1} & A_{74}A_{44}^{-1} & A_{75}S_5^{-1} & A_{76}S_6^{-1} & I \end{array}\right).$$

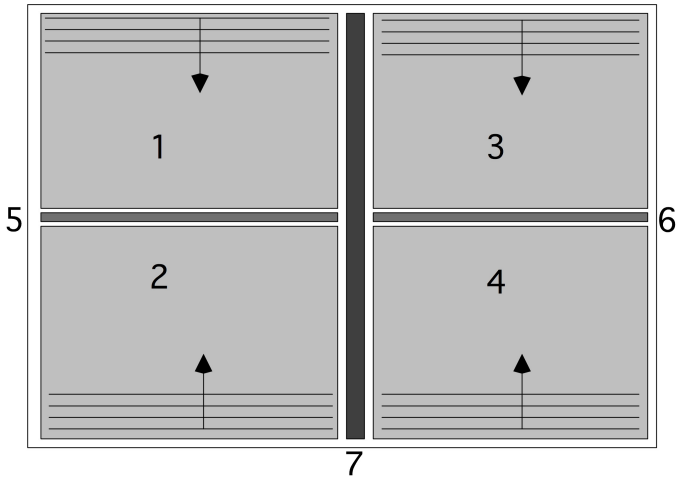


Figure 7.18: A four-way domain decomposition.

$$\left(\begin{array}{cccc|cc|c}
A_{11} & & & & A_{15} & & A_{17} \\
& A_{22} & & & A_{25} & & A_{27} \\
& & A_{33} & & & A_{36} & A_{37} \\
& & & A_{44} & & A_{46} & A_{47} \\
\hline
& & & & S_5 & & A_{57} \\
& & & & & S_6 & A_{67} \\
\hline
& & & & & & S_7
\end{array}\right)$$

where

$$S_5 = A_{55} - A_{51}A_{11}^{-1}A_{15} - A_{52}A_{22}^{-1}A_{25}, \quad S_6 = A_{66} - A_{63}A_{33}^{-1}A_{36} - A_{64}A_{44}^{-1}A_{46}$$
$$S_7 = A_{77} - \sum_{i=1,2,3,4} A_{7i}A_{ii}^{-1}A_{i7} - \sum_{i=5,6} A_{7i}S_i^{-1}A_{17}.$$

Constructing the factorization now goes as follows:

- Blocks A_{ii} are factored in parallel for $i = 1, 2, 3, 4$; similarly the contributions $A_{5i}A_{ii}^{-1}A_{i5}$ for $i = 1, 2$, $A_{6i}A_{ii}^{-1}A_{i6}$ for $i = 3, 4$, and $A_{7i}A_{ii}^{-1}A_{i7}$ for $i = 1, 2, 3, 4$ can be constructed in parallel.
- The Schur complement matrices S_5, S_6 are formed and subsequently factored in parallel, and the contributions $A_{7i}S_i^{-1}A_{17}$ for $i = 5, 6$ are constructed in parallel.
- The Schur complement S_7 is formed and factored.

Analogous to the above reasoning, we conclude that after eliminating blocks 1,2,3,4 the updated matrices S_5, S_6 are dense of size $n/2$, and after eliminating blocks 5,6 the Schur complement S_7 is dense of size n.

Exercise 7.31. Show that solving a system with A^{DD} has a similar parallelism to constructing the factorization as described above.

For future reference, we will call the sets 1 and 2 each others' siblings, and similarly for 3 and 4. The set 5 is the parent of 1 and 2, 6 is the parent of 3 and 4; 5 and 6 are siblings and 7 is the parent of 5 and 6.

7.8.1.1 *Domain decomposition*

In figure 7.18 we divided the domain four ways by a recursive process. This leads up to our discussion of nested dissection. It is also possible to immediately split a domain in any number of strips, or in a grid of subdomains. As long as the separators are wide enough, this will give a matrix structure with many independent subdomains. As in the above discussion, an LU factorization will be characterized by

- parallel processing of the subdomains, both in the factorization and L,U solves, and
- a system to be solved on the separator structure.

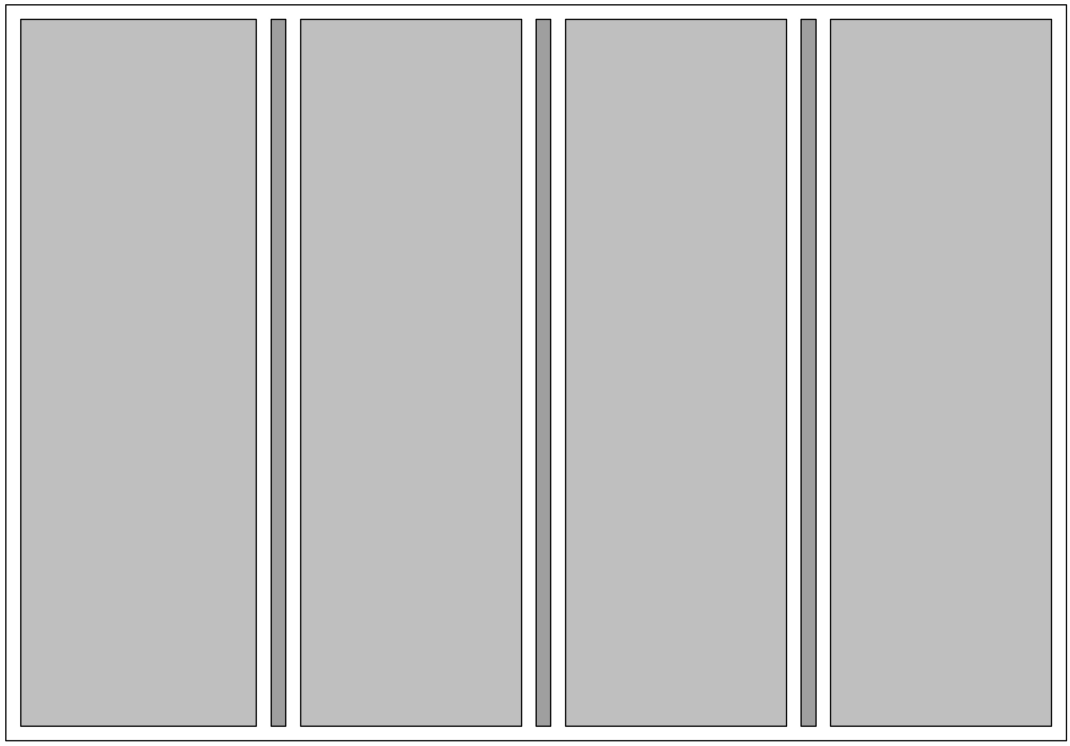


Figure 7.19: One-way domain decomposition.

Exercise 7.32. The matrix from a two-dimensional BVP has a block tridiagonal structure. Divide the domain in four strips, that is, using three separators (see figure 7.19). Note that the separators are uncoupled in the original matrix.
Now sketch the sparsity structure of the resulting system on the separators are elimination of the subdomains. Show that the system is block tridiagonal.

In all the domain splitting schemes we have discussed so far we have used domains that were rectangular, or 'brick' shaped, in more than two dimensions. All of these arguments are applicable to more general domains in two or three dimensions, but things like finding a separator become much harder [140], and that holds even more for the parallel case. See section 20.6.2 for some introduction to this topic.

7.8.1.2 Complexity

The nested dissection method repeats the above process until the subdomains are very small. For a theoretical analysis, we keep dividing until we have subdomains of size 1×1, but in practice one could stop at sizes such as 32, and use an efficient dense solver to factor and invert the blocks.

To derive the complexity of the algorithm, we take another look at figure 7.18, and see that complexity argument, the total space a full recursive nested dissection factorization needs is the sum of

- one dense matrix on a separator of size n, plus
- two dense matrices on separators of size $n/2$,
- taking together $3/2\,n^2$ space and $5/12\,n^3$ time;
- the two terms above then get repeated on four subdomains of size $(n/2) \times (n/2)$.

With the observation that $n = \sqrt{N}$, this sums to

$$\begin{aligned} \text{space} &= 3/2n^2 + 4 \cdot 3/2(n/2)^2 + \cdots \\ &= N(3/2 + 3/2 + \cdots) \quad \log n \text{ terms} \\ &= O(N \log N) \end{aligned}$$

$$\begin{aligned} \text{time} &= 5/12n^3/3 + 4 \cdot 5/12(n/2)^3/3 + \cdots \\ &= 5/12N^{3/2}(1 + 1/4 + 1/16 + \cdots) \\ &= O(N^{3/2}) \end{aligned}$$

Apparently, we now have a factorization that is parallel to a large extent, and that is done in $O(N \log N)$ space, rather than $O(N^{3/2})$ (see section 5.4.3.3). The factorization time has also gone down from $O(N^2)$ to $O(N^{3/2})$.

Unfortunately, this space savings only happens in two dimensions: in three dimensions we need

- one separator of size $n \times n$, taking $(n \times n)^2 = N^{4/3}$ space and $1/3 \cdot (n \times n)^3 = 1/3 \cdot N^2$ time,
- two separators of size $n \times n/2$, taking $N^{3/2}/2$ space and $1/3 \cdot N^2/4$ time,
- four separators of size $n/2 \times n/2$, taking $N^{3/2}/4$ space and $1/3 \cdot N^2/16$ time,
- adding up to $7/4N^{3/2}$ space and $21/16N^2/3$ time;
- on the next level there are 8 subdomains that contribute these terms with $n \to n/2$ and therefore $N \to N/8$.

This makes the total space

$$\frac{7}{4}N^{3/2}(1 + (1/8)^{4/3} + \cdots) = O(N^{3/2})$$

and the total time

$$\frac{21}{16}N^2(1 + 1/16 + \cdots)/3 = O(N^2).$$

We no longer have the tremendous savings of the 2D case.

7.8.1.3 Irregular domains

So far, we have analyzed nested dissection for Cartesian domains. A much more complicated analysis shows that the order improvement holds for general problems in 2D, and that 3D in general has a higher complexity [140].

For irregular domains questions such as how to find the separator become more involved. For this, the concept of a '2D domain' needs to be generalized to that of a *planar graph*. Sufficiently 'good' separators can be found in such graphs [141], but with sequential algorithms. Parallel algorithms rely on *spectral graph theory*; see and 20.6.

7.8.1.4 Parallelism

The nested dissection method clearly introduces a lot of parallelism, and we can characterize it as task parallelism (section 2.5.3): associated with each separator is a task of factoring its matrix, and later one of solving a linear system on its variables. However, the tasks are not independent: in figure 7.18 the factorization on domain 7 has to wait for 5 and 6, and they have to wait for 1,2,3,4. Thus, we have tasks with dependencies in the form of a tree: each separator matrix can be factored only when its children have been factored.

Mapping these tasks to processors is not trivial. First of all, if we are dealing with shared memory we can use a simple task queue:

```
Queue ← {}
for all bottom level subdomains d do
    add d to the Queue
while Queue is not empty do
    if a processor is idle then
        assign a queued task to it
    if a task is finished AND its sibling is finished then
        add its parent to the queue
```

The main problem here is that at some point we will have more processors than tasks, thus causing load unbalance. This problem is made more severe by the fact that the last tasks are also the most substantial, since the separators double in size from level to level. (Recall that the work of factoring a dense matrix goes up with the third power of the size!) Thus, for the larger separators we have to switch from task parallelism to medium-grained parallelism, where processors collaborate on factoring a block.

With distributed memory, we can now solve the parallelism problem with a simple task queue, since it would involve moving large amounts of data. (But recall that work is a higher power of the matrix size, which this time works in our favor, making communication relatively cheap.) The solution is then to use some form of domain decomposition. In figure 7.18 we could have four processors, associated with block 1,2,3,4. Processors 1 and 2 would then negotiate which one factors block 5 (and similarly processors 3 and 4 and block 6), or they could both do it redundantly.

7.8.1.5 Preconditioning

As with all factorizations, it is possible to turn the nested dissection method into a preconditioner by making the factorization incomplete. (For the basic idea of incomplete factorizations, see section 5.5.6.1).



Figure 7.20: Red-black ordering of a the points on a line.

However, here the factorization is formulated completely in terms of *block matrices*, and the division by the pivot element becomes an inversion or system solution with the pivot block matrix. We will not go into this further; for details see the literature [5, 56, 151].

7.8.2 Variable reordering and coloring: independent sets

Parallelism can be achieved in sparse matrices by using graph coloring (section 20.3). Since a 'color' is defined as points that are only connected to other colors, they are by definition independent of each other, and therefore can be processed in parallel. This leads us to the following strategy:

1. Decompose the adjacency graph of the problem into a small number of independent sets, called 'colors';
2. Solve the problem in a number of sequential steps equal to the number of colors; in each step there will be large number of independently processable points.

7.8.2.1 *Red-black coloring*

We start with a simple example, where we consider a tridiagonal matrix A. The equation $Ax = b$ looks like

$$\begin{pmatrix} a_{11} & a_{12} & & & & \emptyset \\ a_{21} & a_{22} & a_{23} & & & \\ & a_{32} & a_{33} & a_{34} & & \\ \emptyset & & \ddots & \ddots & \ddots & \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \end{pmatrix}$$

We observe that x_i directly depends on x_{i-1} and x_{i+1}, but not x_{i-2} or x_{i+1}. Thus, let us see what happens if we permute the indices to group every other component together.

Pictorially, we take the points $1, \ldots, n$ and color them red and black (figure 7.20), then we permute them to first take all red points, and subsequently all black ones. The correspondingly permuted matrix looks as follows:

$$\begin{pmatrix} a_{11} & & & & a_{12} & & \\ & a_{33} & & & a_{32} & a_{34} & \\ & & a_{55} & & & \ddots & \ddots \\ & & & \ddots & & & \\ a_{21} & a_{23} & & & a_{22} & & \\ & a_{43} & a_{45} & & & a_{44} & \\ & & \ddots & \ddots & & & \ddots \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_5 \\ \vdots \\ x_2 \\ x_4 \\ \vdots \end{pmatrix} = \begin{pmatrix} y_1 \\ y_3 \\ y_5 \\ \vdots \\ y_2 \\ y_4 \\ \vdots \end{pmatrix}$$

With this permuted A, the Gauss-Seidel matrix $D_A + L_A$ looks like

$$\begin{pmatrix} a_{11} & & & & \emptyset & & \\ & a_{33} & & & & & \\ & & a_{55} & & & & \\ & & & \ddots & & & \\ a_{21} & a_{23} & & & a_{22} & & \\ & a_{43} & a_{45} & & & a_{44} & \\ & & \ddots & \ddots & & & \ddots \end{pmatrix}$$

What does this buy us? Well, let's spell out the solution of a system $Lx = y$.

for $i = 1, 3, 5, \ldots$ **do**
  solve $x_i \leftarrow y_i / a_{ii}$
for $i = 2, 4, 6, \ldots$ **do**
  compute $t = a_{ii-1}x_{i-1} + a_{ii+1}x_{i+1}$
  solve $x_i \leftarrow (y_i - t)/a_{ii}$

Apparently the algorithm has three stages that are each parallel over half the domain points. This is illustrated in figure 7.21. Theoretically we could accommodate a number of processors that is half the

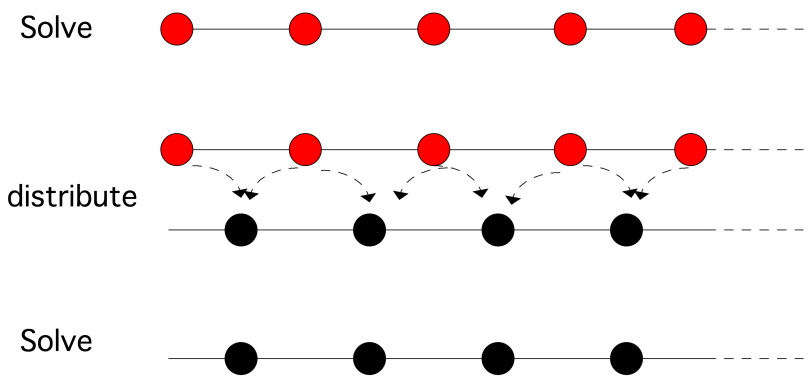


Figure 7.21: Red-black solution on a 1d domain.

number of the domain points, but in practice each processor will have a subdomain. Now you can see in figure 7.22 how this causes a very modest amount of communication: each processor sends at most the data of two red points to its neighbors.

Exercise 7.33. Argue that the adjacency graph here is a *bipartite graph*. We see that such graphs (and in general, colored graphs) are associated with parallelism. Can you also point out performance benefits for non-parallel processors?

Red-black ordering can be applied to two-dimensional problems too. Let us apply a red-black ordering to the points (i, j) where $1 \leq i, j \leq n$. Here we first apply a successive numbering to the odd points on the first line $(1, 1), (3, 1), (5, 1), \ldots$, then the even points of the second line $(2, 2), (4, 2), (6, 2), \ldots$, the odd points on the third line, et cetera. Having thus numbered half the points in the domain, we continue with the even points in the first line, the odd points in the second, et cetera. As you can see in figure 7.23, now the

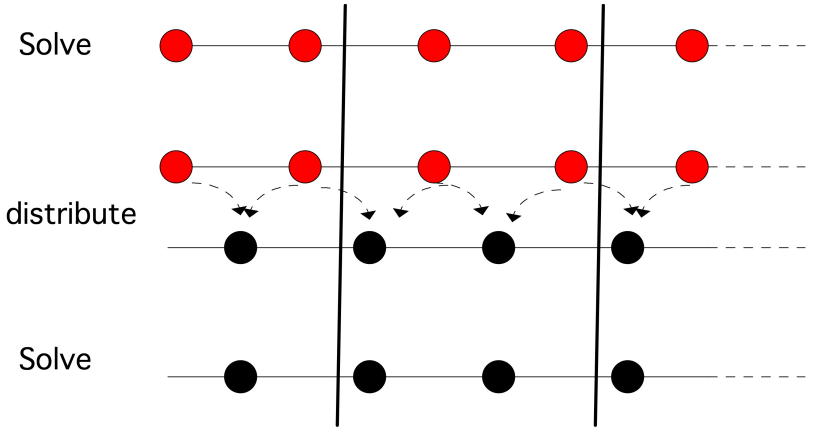


Figure 7.22: Parallel red-black solution on a 1d domain.

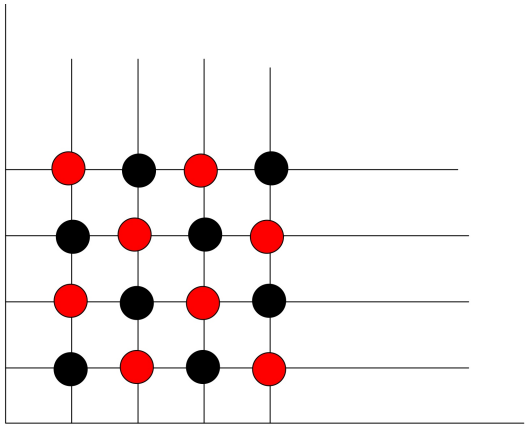


Figure 7.23: Red-black ordering of the variables of a two-dimensional domain.

red points are only connected to black points, and the other way around. In graph theoretical terms, you have found a *coloring* (see appendix 20 for the definition of this concept) of the matrix graph with two colors.

Exercise 7.34. Apply the red-black ordering to the 2D BVP (4.51). Sketch the resulting matrix structure.

The red-black ordering is a simple example of *graph coloring* (sometimes called *multi-coloring*). In simple cases, such as the unit square domain we considered in section 4.2.3 or its extension to 3D, the *color number* of the adjacency graph is readily determined; in less regular cases it is harder.

Exercise 7.35. You saw that a red-black ordering of unknowns coupled with the regular five-point star stencil give two subsets of variables that are not connected among themselves, that is, they form a two-coloring of the matrix graph. Can you find a coloring if nodes are connected by the second stencil in figure 4.3?

7.8.2.2 General coloring

There is a simple bound for the number of colors needed for the graph of a sparse matrix: the number of colors is at most $d+1$ where d is the degree of the graph. To see that we can color a graph with degree d using $d+1$ colors, consider a node with degree d. No matter how its neighbors are colored, there is always an unused color among the $d+1$ available ones.

Exercise 7.36. Consider a sparse matrix, where the graph can be colored with d colors. Permute the matrix by first enumerating the unknowns of the first color, then the second color, et cetera. What can you say about the sparsity pattern of the resulting permuted matrix?

If you are looking for a direct solution of the linear system you can repeat the process of coloring and permuting on the matrix that remains after you have eliminated one color. In the case of a tridiagonal matrix you saw that this remaining matrix was again tridiagonal, so it is clear how to continue the process. This is called *recursive doubling*. If the matrix is not tridiagonal but *block tridiagonal*, this operation can be performed on blocks.

7.8.2.3 Multi-color parallel ILU

In section 7.8.2 you saw the combination of *graph coloring* and permutation. Let P be the permutation that groups like-colored variables together, then $\tilde{A} = P^t AP$ is a matrix with the following structure:

- $\tilde{A}$ has a block structure with the number of blocks equal to the number of colors in the adjacency graph of A; and
- each diagonal block is a diagonal matrix.

Now, if you are performing an iterative system solution and you are looking for a parallel preconditioner you can use this permuted matrix. Consider solving $Ly = x$ with the permuted system. We write the usual algorithm (section 5.3.4) as

for c in the set of colors:
  for i in the variables of color c:
    $y_i \leftarrow x_i - \sum_{j<i} \ell_{ij} y_j$

Exercise 7.37. Show that the flop count of solving a system $LUx = y$ remains the same (in the highest order term) when you from an ILU factorization in the natural ordering to one in the color-permuted ordering.

Where does all this coloring get us? Solving is still sequential...Well, it is true that the outer loop over the colors is sequential, but all the points of one color are independent of each other, so they can be solved at the same time. So if we use an ordinary domain partitioning and combine that with a multi-coloring (see figure 7.24), the processors are all active during all the color stages; see figure 7.25. Ok, if you take a close look at that figure you'll see that one processor is not active in the last color. With large numbers of nodes per processor this is unlikely to happen, but there may be some load imbalance.

One remaining problem is how to generate the multi-coloring in parallel. Finding the optimal color number is NP-hard. The thing that saves us is that we don't necessarily need the optimal number because we are making in incomplete factorization anyway. (There is even an argument that using a slightly larger number of colors decreases the number of iterations.)

An elegant algorithm for finding a multi-coloring in parallel was found by [115, 144]:

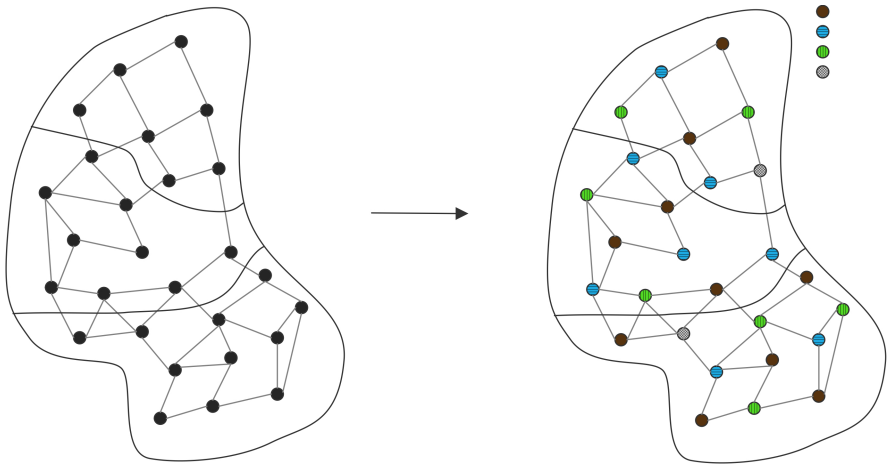


Figure 7.24: A partitioned domain with colored nodes.

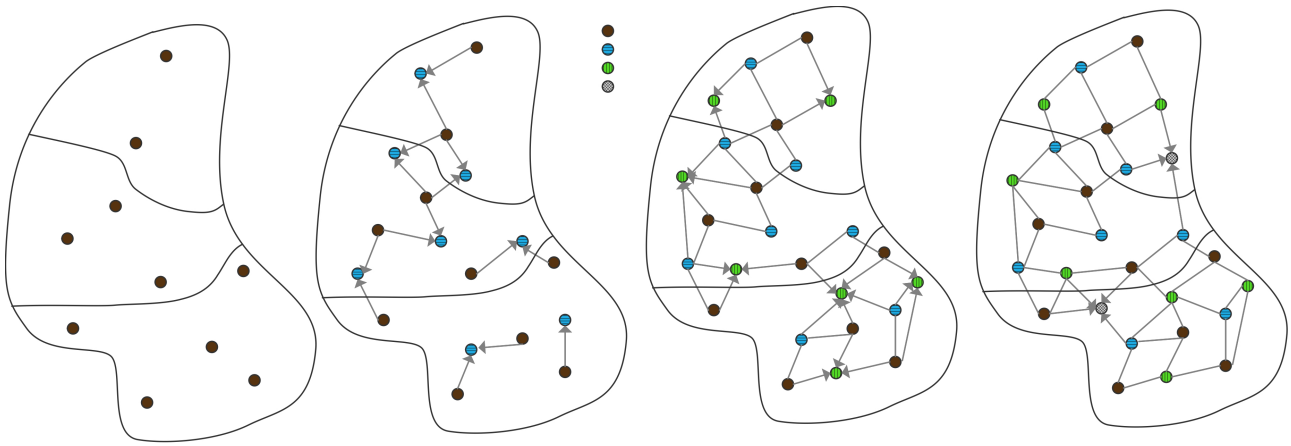


Figure 7.25: Solving a parallel multicolor ILU in four steps.

1. Assign a random value to each variable.
2. Find the variables that have a higher random value than all their neighbors; that is color 1.
3. Then find the variables that have a higher random value than all their neighbors that are not of color 1. That is color 2.
4. Iterate until all points are colored.

7.8.3 Irregular iteration spaces

Applying a computational *stencil*, in the context of *explicit time-stepping* or as a sparse matrix-vector product, is parallel. However, in practice it may not be trivial to split the iteration space. If the iteration space is a Cartesian brick, it is easy, even with nested parallelism.

However, in practice, iterations spaces can be much more complicated. In such cases the following can be done:

1. The iteration loop is traversed to count the total number of inner iterations; this is then divide in as many parts as there processes.
2. The loop is traversed to find the starting and ending index values for each process.
3. The loop code is then rewritten so that it can run over such a subrange.

For a more general discussion, see section 2.10 about load balancing.

7.8.4 Ordering for cache efficiency

The *performance of stencil operations* is often quite low. The operations have no obvious cache reuse; often they resemble *stream operations* where long streams of data are retrieved from memory and used only once. If only a single stencil evaluation was done, that would be the end of the story. However, usually we do many such updates, and we can apply techniques similar to *loop tiling* described in section 6.7.

We can do better than with plain tiling if we take the shape of the stencil into account. Figure 7.26 shows

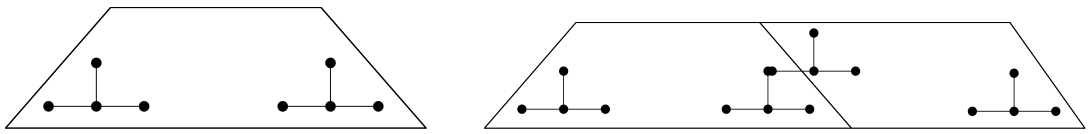


Figure 7.26: First and second step in cache-optimized iteration space traversal.

(left) how we first compute a time-space trapezoid that is cache-contained. Then (right) we compute another cache-contained trapezoid that builds on the first [69].

7.9 Parallelism in solving linear systems from PDEs

The numerical solution of PDEs is an important activity, and the precision required often makes it a prime candidate for parallel treatment. If we wonder just how parallel we can be, and in particular what sort of a speedup is attainable, we need to distinguish between various aspects of the question.

First of all we can ask if there is any intrinsic parallelism in the problem. On a global level this will typically not be the case (if parts of the problem were completely uncoupled, then they would be separate problems, right?) but on a smaller level there may be parallelism.

For instance, looking at *time-dependent problems*, and referring to section 4.2.1, we can say that every next time step is of course dependent on the previous one, but not every individual point on the next time step is dependent on every point in the previous step: there is a *region of influence*. Thus it may be possible to partition the problem domain and obtain parallelism.

In PDEs, we can also make a high-level statement about their *parallel solution* based on their characterization in term of what is known in PDE theory as *characteristics*. A *wave equation* is intuitively a strictly forward going phenomenon, so the next time step is an explicit, parallel computable, function of the previous. On the other hand, *elliptic problems* (section refsec:elliptic) have every point depending on every other one. Therefore a parallel solution is probably going to involve some sequence of more explicit approximations. We will discuss that in the next subsection for a simple case.

7.9.1 The Laplace/Poisson equation on the unit square

The *Laplace equation on the unit square*

$$-\Delta u = 0, \qquad \text{on } \Omega = [0,1]^2 \tag{7.2}$$

is the simplest both-serious-and-useful PDE problem. If we discretize this to second order, using either the FEM or FDM, we obtain the matrix of (4.56).

The following approaches to solving this implicit system exist.

- The *Jacobi iteration*, equation (5.16), is a fully parallel computation over all points, but it takes many sweeps over the domain.
- The closely related *Gauss-Seidel iteration* converges faster, but is implicit. The common way to parallelize it is by *multi-coloring*; section 7.8.2. Another approach is possible with *wavefronts*; section 7.10.1.
- A much better iteration scheme is to use the *CG* method; section 5.5.11. This is typically accelerated by a preconditioner, for which often some variant of the above Jacobi or Gauss-Seidel iteration is chosen; see sections 5.5.6 and 7.7.
- Finally, simple equations like the Laplace equation can be solved by *multigrid* methods, whether by themselves or as preconditioner for an iterative method. We will not discus that in this book.

7.9.2 Operator splitting

In some contexts, it is necessary to perform implicit calculations through all directions of a two or three-dimensional array. For example, in section 4.3 you saw how the implicit solution of the heat equation gave rise to repeated systems

$$(\alpha I + \frac{d^2}{dx^2} + \frac{d^2}{dy^2})u^{(t+1)} = u^{(t)} \tag{7.3}$$

Without proof, we state that the time-dependent problem can also be solved by

$$(\beta I + \frac{d^2}{dx^2})(\beta I + \frac{d^2}{dy^2})u^{(t+1)} = u^{(t)} \tag{7.4}$$

for suitable β. This scheme will not compute the same values on each individual time step, but it will converge to the same steady state. The scheme can also be used as a preconditioner in the BVP case.

This approach has considerable advantages, mostly in terms of operation counts: the original system has to be solved either making a factorization of the matrix, which incurs *fill-in*, or by solving it iteratively.

Exercise 7.38. Analyze the relative merits of these approaches, giving rough operation counts. Consider both the case where α has dependence on t and where it does not. Also discuss the expected speed of various operations.

A further advantage appears when we consider the parallel solution of (7.4). Note that we have a two-dimensional set of variables u_{ij}, but the operator $I + d^2u/dx^2$ only connects $u_{ij}, u_{ij-1}, u_{ij+1}$. That is, each line corresponding to an i value can be processed independently. Thus, both operators can be solved fully parallel using a one-dimensional partition on the domain. The solution of the system in (7.3), on the other hand, has limited parallelism.

Unfortunately, there is a serious complication: the operator in x direction needs a partitioning of the domain in on direction, and the operator in y in the other. The solution usually taken is to transpose the u_{ij} value matrix in between the two solves, so that the same processor decomposition can handle both. This transposition can take a substantial amount of the processing time of each time step.

Exercise 7.39. Discuss the merits of and problems with a two-dimensional decomposition of the domain, using a grid of $P = p \times p$ processors. Can you suggest a way to ameliorate the problems?

One way to speed up these calculations, is to replace the implicit solve, by an explicit operation; see section 7.10.3.

7.10 Parallelism and implicit operations

In the discussion of IBVPs (section 4.1.2.2) you saw that implicit operations can have great advantages from the point of numerical stability. However, you also saw that they make the difference between methods based on a simple operation such as the matrix-vector product, and ones based on the more complicated linear system solution. There are further problems with implicit methods when you start computing in parallel.

The following exercise serves as a recap of some of the issues already identified in our general discussion of linear algebra; chapter 5.

Exercise 7.40. Let A be the matrix

$$A = \begin{pmatrix} a_{11} & & \emptyset & \\ a_{21} & a_{22} & & \\ & \ddots & \ddots & \\ \emptyset & & a_{n,n-1} & a_{nn} \end{pmatrix}. \tag{7.5}$$

Show that the matrix vector product $y \leftarrow Ax$ and the system solution $x \leftarrow A^{-1}y$, obtained by solving the triangular system $Ax = y$, not by inverting A, have the same operation count.
Now consider parallelizing the product $y \leftarrow Ax$. Suppose we have n processors, and each processor i stores x_i and the i-th row of A. Show that the product Ax can be computed without idle time on any processor but the first.
Can the same be done for the solution of the triangular system $Ax = y$? Show that the straightforward implementation has every processor idle for an $(n-1)/n$ fraction of the computation.

We will now see a number of ways of dealing with this inherently sequential component.

7.10.1 Wavefronts

Above, you saw that solving a lower triangular system of size N can have sequential time complexity of N steps. In practice, things are often not quite that bad. Implicit algorithms such as solving a triangular system are inherently sequential, but the number of steps can be less than is apparent at first.

Exercise 7.41. Take another look at the matrix (4.56) from a two-dimensional BVP on the unit square, discretized with central differences. This matrix resulted from the *lexicographic ordering* of the variables. Derive the matrix structure if we order the unknowns by diagonals. What can you say about the sizes of the blocks and the structure of the blocks themselves? What is the implication for parallelism?

We don't have to form the matrix to arrive at a parallelism analysis. As already remarked, often a stencil approach is more fruitful. Let us take another look at figure 4.1 that describes the *finite difference stencil* of a two-dimensional BVP. The corresponding picture for the *stencil of the lower triangular factor* is in figure 7.27. This describes the sequentiality of the lower triangular solve process $x \leftarrow L^{-1}y$:

$$x_k = y_k - \ell_{k,k-1}x_{k-1} - \ell_{k,k-n}x_{k-n}$$

In other words, the value at point k can be found if its neighbors to the left (that is, variable $k-1$) and below (variable $k-n$) are known.

Turning this around, we see that, if we know x_1, we can not only find x_2, but also x_{n+1}. In the next step we can determine x_3, x_{n+2}, and x_{2n+1}. Continuing this way, we can solve x by *wavefronts*: the values of x on each wavefront are independent, so they can be solved in parallel in the same sequential step.

Exercise 7.42. Finish this argument. What is the maximum number of processors we can employ, and what is the number of sequential steps? What is the resulting efficiency?

Of course you don't have to use actual parallel processing to exploit this parallelism. Instead you could use a *vector processor*, *vector instructions*, or a *GPU* [143].

In section 5.4.3.5 you saw the *Cuthill-McKee ordering* for reducing the fill-in of a matrix. We can modify this algorithm as follows to give wavefronts:

1. Take an arbitrary node, and call that 'level zero'.
2. For level $n+1$, find points connected to level n, that are not themselves connected.
3. For the so-called 'reverse Cuthill-McKee ordering', reverse the numbering of the levels.

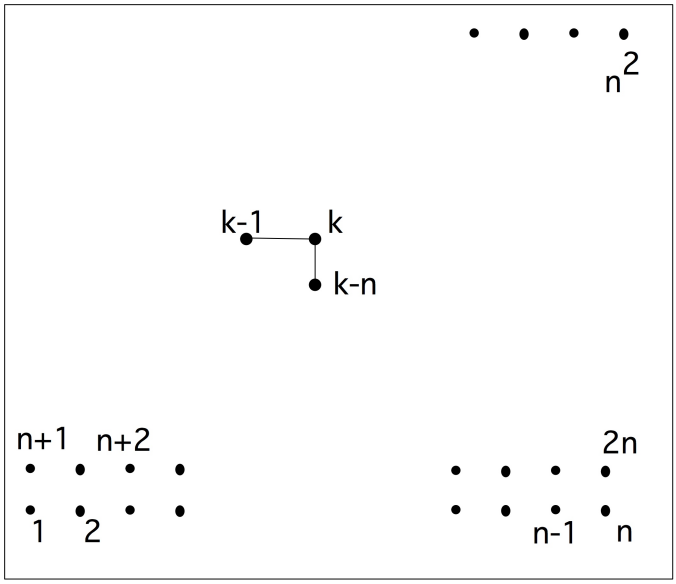


Figure 7.27: The difference stencil of the L factor of the matrix of a two-dimensional BVP.

Exercise 7.43. This algorithm is not entirely correct. What is the problem; how can you correct it? Show that the resulting permuted matrix is no longer tridiagonal, but will likely still have a band structure.

7.10.2 Recursive doubling

Recursions $y_{i+1} = a_i y_i + b_i$, such as appear in solving a bilinear set of equations (see exercise 4.4), seem intrinsically sequential. However, you already saw in exercise 1.4 how, at the cost of some preliminary operations, the computation can be parallelized. The basic idea is that you transform the sequence:

$$x_{i+1} = a_i x_i + y_i \Rightarrow x_{i+2} = a_i' x_i + y_i' \Rightarrow x_{i+4} = a_i'' x_i + y_i'' \Rightarrow \cdots$$

We will now formalize this strategy, generally known as *recursive doubling*, First, take the general *bidiagonal matrix* from (7.5) and scale it to be of the normalized form, giving the problem to solve x in:

$$\begin{pmatrix} 1 & & \emptyset \\ b_{21} & 1 & \\ & \ddots & \ddots \\ \emptyset & & b_{n,n-1} \quad 1 \end{pmatrix} x = y$$

which we write as $A = I + B$.

Exercise 7.44. Show that the scaling to normalized form can be done by multiplying with a diagonal matrix. How does solving the system $(I + B)x = y$ help in solving $Ax = y$? What are the operation counts of solving the system in the two different ways?

Now we do something that looks like Gaussian elimination, except that we do not start with the first row, but the second. (What would happen if you did Gaussian elimination or LU decomposition on the matrix $I + B$?) We use the second row to eliminate b_{32}:

$$\begin{pmatrix} 1 & & & \emptyset \\ & 1 & & \\ & -b_{32} & 1 & \\ & & & \ddots \\ \emptyset & & & 1 \end{pmatrix} \times \begin{pmatrix} 1 & & & \emptyset \\ b_{21} & 1 & & \\ & b_{32} & 1 & \\ & & \ddots & \ddots \\ \emptyset & & b_{n,n-1} & 1 \end{pmatrix} = \begin{pmatrix} 1 & & & \emptyset \\ b_{21} & 1 & & \\ -b_{32}b_{21} & 0 & 1 & \\ \emptyset & & b_{n,n-1} & 1 \end{pmatrix}$$

which we write as $L^{(2)}A = A^{(2)}$. We also compute $L^{(2)}y = y^{(2)}$ so that $A^{(2)}x = y^{(2)}$ has the same solution as $Ax = y$. Solving the transformed system gains us a little: after we compute x_1, x_2 and x_3 can be computed in parallel.

Now we repeat this elimination process by using the fourth row to eliminate b_{54}, the sixth row to eliminate b_{76}, et cetera. The final result is, summarizing all $L^{(i)}$ matrices:

$$\begin{pmatrix} 1 & & & & & & & \emptyset \\ 0 & 1 & & & & & & \\ & -b_{32} & 1 & & & & & \\ & & 0 & 1 & & & & \\ & & & -b_{54} & 1 & & & \\ & & & & 0 & 1 & & \\ & & & & & -b_{76} & 1 & \\ & & & & & & \ddots & \ddots \end{pmatrix} \times (I + B) = \begin{pmatrix} 1 & & & & & & & \emptyset \\ b_{21} & 1 & & & & & & \\ -b_{32}b_{21} & 0 & 1 & & & & & \\ & & b_{43} & 1 & & & & \\ & & -b_{54}b_{43} & 0 & 1 & & & \\ & & & & b_{65} & 1 & & \\ & & & & -b_{76}b_{65} & 0 & 1 & \\ & & & & & \ddots & \ddots & \ddots \end{pmatrix}$$

which we write as $L(I + B) = C$, and solving $(I + B)x = y$ now becomes $Cx = L^{-1}y$.

This final result needs close investigation.

- First of all, computing $y' = L^{-1}y$ is simple. (Work out the details. How much parallelism is available?)
- Solving $Cx = y'$ is still sequential, but it no longer takes n steps: from x_1 we can get x_3, from that we get x_5, et cetera. In other words, there is only a sequential relationship between the odd numbered components of x.
- The even numbered components of x do not depend on each other, but only on the odd components: x_2 follows from x_1, x_4 from x_3, et cetera. Once the odd components have been computed, admittedly sequentially, this step is fully parallel.

We can describe the sequential solving of the odd components by itself:

$$\begin{pmatrix} 1 & & \emptyset & \\ c_{31} & 1 & & \\ & \ddots & \ddots & \\ \emptyset & & c_{n,n-1} & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y'_1 \\ y'_3 \\ \vdots \\ y'_n \end{pmatrix}$$

where $c_{i+1i} = -b_{2n+1,2n}b_{2n,2n-1}$. In other words, we have reduced a size n sequential problem to a sequential problem of the same kind and a parallel problem, both of size $n/2$. Now we can repeat this procedure

recursively, reducing the original problem to a sequence of parallel operations, each half the size of the former.

The process of computing all partial sums through recursive doubling is also referred to as a parallel *prefix operation*. Here we use a prefix sum, but in the abstract it can be applied to any associative operator; see section 22.

7.10.3 Approximating implicit by explicit operations, series expansion

There are various reasons why it is sometimes allowed to replace an implicit operation, which, as you saw above, can be problematic in practice, by a different one that is practically more advantageous.

- Using an explicit method for the heat equation (section 4.3) instead of an implicit one is equally legitimate, as long as we observe *step size* restrictions on the explicit method.
- Tinkering with the preconditioner (section 5.5.8) in an iterative method is allowed, since it will only affect the speed of convergence, not the solution the method converges to. You already saw one example of this general idea in the *block Jacobi* method; section 7.7.4. In the rest of this section you will see how recurrences in the preconditioner, which are implicit operations, can be replaced by explicit operations, giving various computational advantages.

Solving a linear system is a good example of an implicit operation, and since this comes down to solving two triangular systems, let us look at ways of finding a computational alternative to solving a lower triangular system. If U is upper triangular and nonsingular, we let D be the diagonal of U, and we write $U = D(I - B)$ where B is an upper triangular matrix with a zero diagonal, also called a *strictly upper triangular matrix*; we say that $I - B$ is a *unit upper triangular matrix*.

Exercise 7.45. Let $A = LU$ be an LU factorization where L has ones on the diagonal, and the pivots on the diagonal of U, as derived in section 5.3. Show how solving a system $Ax = b$ can be done, involving only the solution of unit upper and lower triangular systems. Show that no divisions are needed during the system solution.

Our operation of interest is now solving the system $(I - B)x = y$. We observe that

$$(I - B)^{-1} = I + B + B^2 + \cdots \tag{7.6}$$

and $B^n = 0$ where n is the matrix size (check this!), so we can solve $(I - B)x = y$ exactly by

$$x = \sum_{k=0}^{n-1} B^k y.$$

Of course, we want to avoid computing the powers B^k explicitly, so we observe that

$$\begin{aligned} x_1 = \textstyle\sum_{k=0}^{1} B^k y &= (I + B)y, \\ x_2 = \textstyle\sum_{k=0}^{2} B^k y &= (I + B(I + B))y = y + Bx_1, \\ x_3 = \textstyle\sum_{k=0}^{3} B^k y &= (I + B(I + B((I + B))))y = y + Bx_2 \end{aligned} \tag{7.7}$$

et cetera. The resulting algorithm for evaluating $\sum_{k=0}^{n-1} B^k y$ is called *Horner's rule* (see section 22.3), and you see that it avoids computing matrix powers B^k.

Exercise 7.46. Suppose that $I-B$ is bidiagonal. Show that the above calculation takes $n(n+1)$ operations. (What would it have been if you'd computed the powers of B explicitly?) What is the operation count for computing $(I-B)x=y$ by triangular solution?

We have now turned an implicit operation into an explicit one, but unfortunately one with a high operation count. In practical circumstances, as argued above, however, we can often justify truncating the sum of matrix powers.

Exercise 7.47. Let A be the tridiagonal matrix

$$A=\begin{pmatrix} 2 & -1 & & & \emptyset \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & & & -1 \\ \emptyset & & & -1 & 2 \end{pmatrix}$$

of the one-dimensional BVP from section 4.2.2.

1. Recall the definition of diagonal dominance in section 5.3.3. Is this matrix diagonally dominant?
2. Show that the pivots in an LU factorization of this matrix (without pivoting) satisfy a recurrence. Hint: show that after n elimination steps ($n\geq 0$) the remaining matrix looks like

$$A^{(n)}=\begin{pmatrix} d_n & -1 & & & \emptyset \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & & & -1 \\ \emptyset & & & -1 & 2 \end{pmatrix}$$

   and show the relation between d_{n+1} and d_n.
3. Show that the sequence $n\mapsto d_n$ is descending, and derive the limit value.
4. Write out the L and U factors in terms of the d_n pivots.
5. Are the L and U factors diagonally dominant?

The above exercise implies (note that we did not actually prove it!) that for matrices from BVPs we find that $B^k\downarrow 0$, in element size and in norm. This means that we can approximate the solution of $(I-B)x=y$ by, for instance, $x=(I+B)y$ or $x=(I+B+B^2)y$. Doing this still has a higher operation count than the direct triangular solution, but it is computationally advantageous in at least two ways:

- The explicit algorithm has a better pipeline behavior.
- The implicit algorithm has problems in parallel, as you have seen; the explicit algorithm is more easily parallelized.

See also [187]. Of course, this approximation may have further implications for the stability of the overall numerical algorithm.

Exercise 7.48. Describe the parallelism aspects of Horner's rule; equation (7.7).

7.11 Grid updates

One of the conclusions of chapter 4 was that explicit methods for time-dependent problems are computationally easier than implicit ones. For instance, they typically involve a matrix-vector product rather than a system solution, and parallelizing explicit operations is fairly simple: each result value of the matrix-vector product can be computed independently. That does not mean that there are no other computational aspects worth remarking on.

Since we are dealing with sparse matrices, stemming from some computational *stencil*, we take the operator point of view. In figures 7.12 and 7.13 you saw how applying a stencil in each point of the domain induces certain relations between processors: in order to evaluate the matrix-vector product $y \leftarrow Ax$ on a processor, that processor needs to obtain the x-values of its *halo region*. Under reasonable assumptions on the partitioning of the domain over the processors, the number of messages involved will be fairly small.

Exercise 7.49. Reason that, in a FEM or FDM context, the number of messages is $O(1)$ as $h \downarrow 0$.

In section 1.6.1 you saw that the matrix-vector product has little data reuse, though there is some locality to the computation; in section 5.4.1.4 it was pointed out that the locality of the sparse matrix-vector product is even worse because of indexing schemes that the sparsity necessitates. This means that the sparse product is largely a *bandwidth-bound algorithm*.

Looking at just a single product there is not much we can do about that. However, often we do a number of such products in a row, for instance as the steps in a time-dependent process. In that case there may be rearrangements of the operations that lessen the bandwidth demands. Consider as a simple example

$$\forall_i : x_i^{(n+1)} = f\left(x_i^{(n)}, x_{i-1}^{(n)}, x_{i+1}^{(n)}\right) \tag{7.8}$$

and let's assume that the set $\{x_i^{(n)}\}_i$ is too large to fit in cache. This is a model for, for instance, the explicit scheme for the heat equation in one space dimension; section 4.3.1.1. Schematically:

$$\begin{array}{ccc}
x_0^{(n)} & x_1^{(n)} & x_2^{(n)} \\
\downarrow\swarrow & \searrow\downarrow\swarrow & \searrow\downarrow\swarrow \\
x_0^{(n+1)} & x_1^{(n+1)} & x_2^{(n+1)} \\
\downarrow\swarrow & \searrow\downarrow\swarrow & \searrow\downarrow\swarrow \\
x_0^{(n+2)} & x_1^{(n+2)} & x_2^{(n+2)}
\end{array}$$

In the ordinary computation, where we first compute all $x_i^{(n+1)}$, then all $x_i^{(n+2)}$, the intermediate values at level $n+1$ will be flushed from the cache after they were generated, and then brought back into cache as input for the level $n+2$ quantities.

However, if we compute not one, but two iterations, the intermediate values may stay in cache. Consider $x_0^{(n+2)}$: it requires $x_0^{(n+1)}, x_1^{(n+1)}$, which in turn require $x_0^{(n)}, \ldots, x_2^{(n)}$.

Now suppose that we are not interested in the intermediate results, but only the final iteration. This is the case if you are iterating to a stationary state, doing multigrid smoothing, et cetera. Figure 7.28 shows a simple example. The first processor computes 4 points on level $n+2$. For this it needs 5 points from level $n+1$, and these need to be computed too, from 6 points on level n. We see that a processor apparently

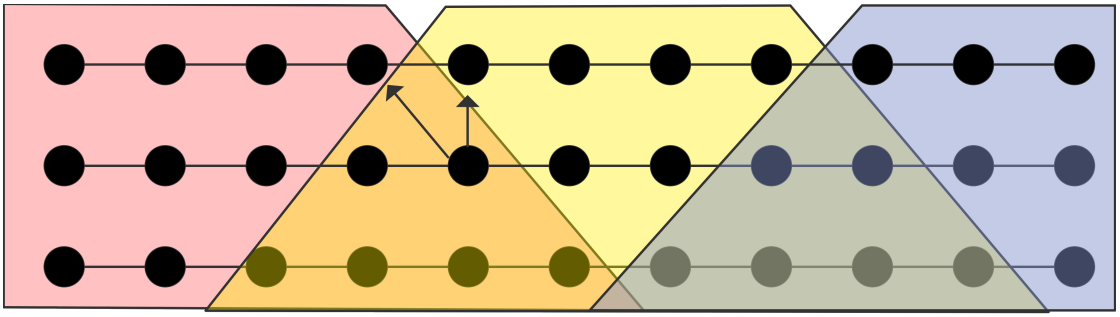


Figure 7.28: Computation of blocks of grid points over multiple iterations.

needs to collect a *ghost region* of width two, as opposed to just one for the regular single step update. One of the points computed by the first processor is $x_3^{(n+2)}$, which needs $x_4^{(n+1)}$. This point is also needed for the computation of $x_4^{(n+2)}$, which belongs to the second processor.

The easiest solution is to let this sort of point on the intermediate level *redundantly computed*, in the computation of both blocks where it is needed, on two different processors.

Exercise 7.50. Can you think of cases where a point would be redundantly computed by more than two processors?

We can give several interpretations to this scheme of computing multiple update steps by blocks.

- First of all, as we motivated above, doing this on a single processor increases locality: if all points in a colored block (see the figure) fit in cache, we get reuse of the intermediate points.
- Secondly, if we consider this as a scheme for distributed memory computation, it reduces message traffic. Normally, for every update step the processors need to exchange their boundary data. If we accept some redundant duplication of work, we can now eliminate the data exchange for the intermediate levels. The decrease in communication will typically outweigh the increase in work.

Exercise 7.51. Discuss the case of using this strategy for multicore computation. What are the savings? What are the potential pitfalls?

7.11.1 Analysis

Let's analyze the algorithm we have just sketched. As in equation (7.8) we limit ourselves to a 1D set of points and a function of three points. The parameters describing the problem are these:

- N is the number of points to be updated, and M denotes the number of update steps. Thus, we perform MN function evaluations.
- α, β, γ are the usual parameters describing latency, transmission time of a single point, and time for an operation (here taken to be an f evaluation).
- b is the number of steps we block together.

Each halo communication consists of b points, and we do this $\sqrt{N}/b$ many times. The work performed consists of the MN/p local updates, plus the redundant work because of the halo. The latter term consists of $b^2/2$ operations, performed both on the left and right side of the processor domain.

Adding all these terms together, we find a cost of

$$\frac{M}{b}\alpha + M\beta + \left(\frac{MN}{p} + Mb\right)\gamma.$$

We observe that the overhead of $\alpha M/b + \gamma Mb$ is independent of p,

Exercise 7.52. Compute the optimal value of b, and remark that it only depends on the architectural parameters α, β, γ but not on the problem parameters.

7.11.2 Communication and work minimizing strategy

We can make this algorithm more efficient by *overlapping computation with communication*. As illustrated in figure 7.29, each processor start by communicating its halo, and overlapping this communication with the part of the communication that can be done locally. The values that depend on the halo will then be computed last.

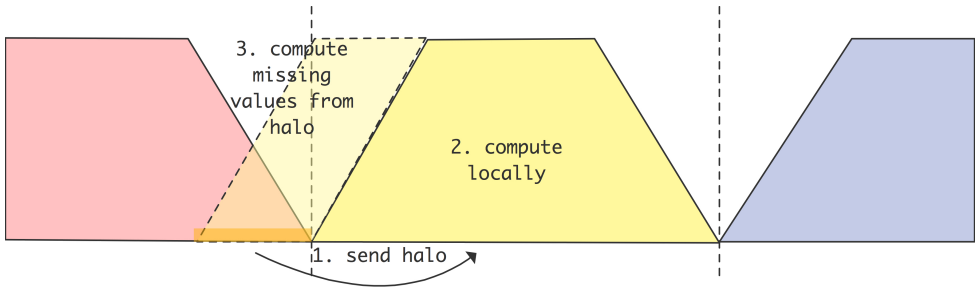


Figure 7.29: Computation of blocks of grid points over multiple iterations.

Exercise 7.53. What is a great practical problem with organizing your code (with the emphasis on 'code'!) this way?

If the number of points per processor is large enough, the amount of communication is low relative to the computation, and you could take b fairly large. However, these grid updates are mostly used in iterative methods such as the *CG* method (section 5.5.11), and in that case considerations of roundoff prevent you from taking b too large[33].

Exercise 7.54. Go through the complexity analysis for the non-overlapping algorithm in case the points are organized in a 2D grid. Assume that each point update involves four neighbors, two in each coordinate direction.

A further refinement of the above algorithm is possible. Figure 7.30 illustrates that it is possible to use a halo region that uses different points from different time steps. This algorithm (see [43]) cuts down on the amount of redundant computation. However, now the halo values that are communicated first need to be computed, so this requires splitting the local communication into two phases.

7.12 Block algorithms on multicore architectures

In section 5.3.6 you saw that certain linear algebra algorithms can be formulated in terms of submatrices. This point of view can be beneficial for the efficient execution of linear algebra operations on shared

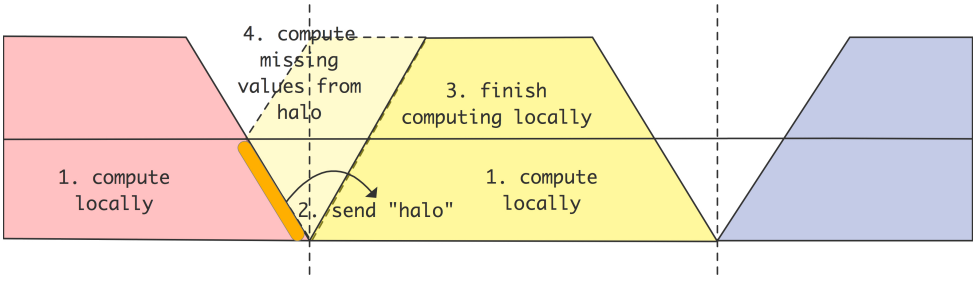


Figure 7.30: Computation of blocks of grid points over multiple iterations.

memory architectures such as current *multicore* processors.

As an example, let us consider the *Cholesky factorization*, which computes $A = LL^t$ for a symmetric positive definite matrix A; see also section 5.3.1.2. Recursively, we can describe the algorithm as follows:

$$\mathrm{Chol}\begin{pmatrix} A_{11} & A_{21}^t \\ A_{21} & A_{22} \end{pmatrix} = LL^t \qquad \text{where} \quad L = \begin{pmatrix} L_{11} & 0 \\ \tilde{A}_{21} & \mathrm{Chol}(A_{22} - \tilde{A}_{21}\tilde{A}_{21}^t) \end{pmatrix}$$

and where $\tilde{A}_{21} = A_{21}L_{11}^{-t}$, $A_{11} = L_{11}L_{11}^t$.

(This algorithm applies can be considered both as a scalar and *block matrix* algorithm. From a point of performance, by using *BLAS 3* routines, we will implicitly consider only the blocked approach.)

In practice, the block implementation is applied to a partitioning

$$\left(\begin{array}{c|cc} & \multicolumn{2}{c}{\text{finished}} \\ \hline & A_{kk} & A_{k,>k} \\ \hline & A_{>k,k} & A_{>k,>k} \end{array}\right)$$

where k is the index of the current block row, and the factorization is finished for all indices $< k$. The factorization is written as follows, using BLAS names for the operations:

for $k = 1$, nblocks:
- `Chol`: factor $L_kL_k^t \leftarrow A_{kk}$
- `Trsm`: solve $\tilde{A}_{>k,k} \leftarrow A_{>k,k}L_k^{-t}$
- `Gemm`: form the product $\tilde{A}_{>k,k}\tilde{A}_{>k,k}^t$
- `Syrk`: symmmetric rank-k update $A_{>k,>k} \leftarrow A_{>k,>k} - \tilde{A}_{>k,k}\tilde{A}_{>k,k}^t$

The key to parallel performance is to partition the indices $> k$ and write the algorithm in terms of these blocks:

$$\left(\begin{array}{c|cc} & \multicolumn{2}{c}{\text{finished}} \\ \hline & A_{kk} & A_{k,k+1} \quad A_{k,k+2}\cdots \\ \hline & A_{k+1,k} & A_{k+1,k+1} \quad A_{k+1,k+2}\cdots \\ & A_{k+2,k} & A_{k+2,k+2} \\ & \vdots & \vdots \end{array}\right)$$

The algorithm now gets an extra level of inner loops:

for $k = 1$, nblocks:
  `Chol`: factor $L_k L_k^t \leftarrow A_{kk}$
  for $\ell > k$:
    `Trsm`: solve $\tilde{A}_{\ell,k} \leftarrow A_{\ell,k} L_k^{-t}$
  for $\ell_1, \ell_2 > k$:
    `Gemm`: form the product $\tilde{A}_{\ell_1,k} \tilde{A}_{\ell_2,k}^t$
  for $\ell_1, \ell_2 > k$, $\ell_1 \leq \ell_2$:
    `Syrk`: symmmetric rank-k update $A_{\ell_1,\ell_2} \leftarrow A_{\ell_1,\ell_2} - \tilde{A}_{\ell_1,k} \tilde{A}_{\ell_2,k}^t$

Now it is clear that the algorithm has a good deal of parallelism: the iterations in every ℓ-loop can be processed independently. However, these loops get shorter in every iteration of the outer k-loop, so it is not immediate how many processors we can accommodate. On the other hand, it is not necessary to preserve the order of operations of the algorithm above. For instance, after

$$L_1 L_1^t = A_{11}, \quad A_{21} \leftarrow A_{21} L_1^{-t}, \quad A_{22} \leftarrow A_{22} - A_{21} A_{21}^t$$

the factorization $L_2 L_2^t = A_{22}$ can start, even if the rest of the $k = 1$ iteration is still unfinished. Thus, there is probably a lot more parallelism than we would get from just parallelizing the inner loops.

The best way to approach parallelism in this case is to shift away from a *control flow* view of the algorithm, where the sequence of operations is prescribed, to a *data flow* view. In the latter only data dependencies are indicated, and any ordering of operations that obeys these dependencies is allowed. (Technically, we abandon the program order of the tasks and replace it with a *partial ordering*[4] .) The best way of representing the data flow of an algorithm is by constructing a *Directed Acyclic Graph (DAG)* (see section 20 for a brief tutorial on graphs) of tasks. We add an edge (i, j) to the graph if task j uses the output of task i.

Exercise 7.55. In section 2.6.1.6 you learned the concept of *sequential consistency*: a threaded parallel code program should give the same results when executed in parallel as when it's executed sequentially. We have just stated that DAG-based algorithms are free to execute tasks in any order that obeys the partial order of the graph nodes. Discuss whether sequential consistency is a problem in this context.

In our example, we construct a DAG by making a vertex task for every inner iteration. Figure 7.31 shows the DAG of all tasks of matrix of 4×4 blocks. This graph is constructed by simulating the Cholesky algorithm above,

Exercise 7.56. What is the diameter of this graph? Identify the tasks that lie on the path that determines the diameter. What is the meaning of these tasks in the context of the algorithm?

This path is called the *critical path*. Its length determines the execution time of the computation in parallel, even if an infinite number of processors is available. (These topics were already discussed in section 2.2.4.)

Exercise 7.57. Assume there are T tasks that all take a unit time to execute, and assume we have p processors. What is the theoretical minimum time to execute the algorithm? Now amend this formula to take into account the critical path; call its length C.

4. Let's write $a \leq b$ if a is executed before b, then the relation $\cdot \leq \cdot$ is a partial order if $a \leq b \wedge b \leq a \Rightarrow a = b$ and $a \leq b \wedge b \leq c \Rightarrow a \leq c$. The difference with a total ordering, such as program ordering, is that it is not true that $a \leq b \vee b \leq a$: there can be pairs that are not ordered, meaning that their time ordering is not prescribed.

In the execution of the tasks a DAG, several observations can be made.

- If more than one update is made to a block, it is probably advantageous to have these updates be computed by the same process. This simplifies maintaining *cache coherence*.
- If data is used and later modified, the use must be finished before the modification can start. This can even be true if the two actions are on different processors, since the memory subsystem typically maintains cache coherence, so the modifications can affect the process that is reading the data. This case can be remedied by having a copy of the data in main memory, giving a reading process data that is reserved (see section 1.4.1).

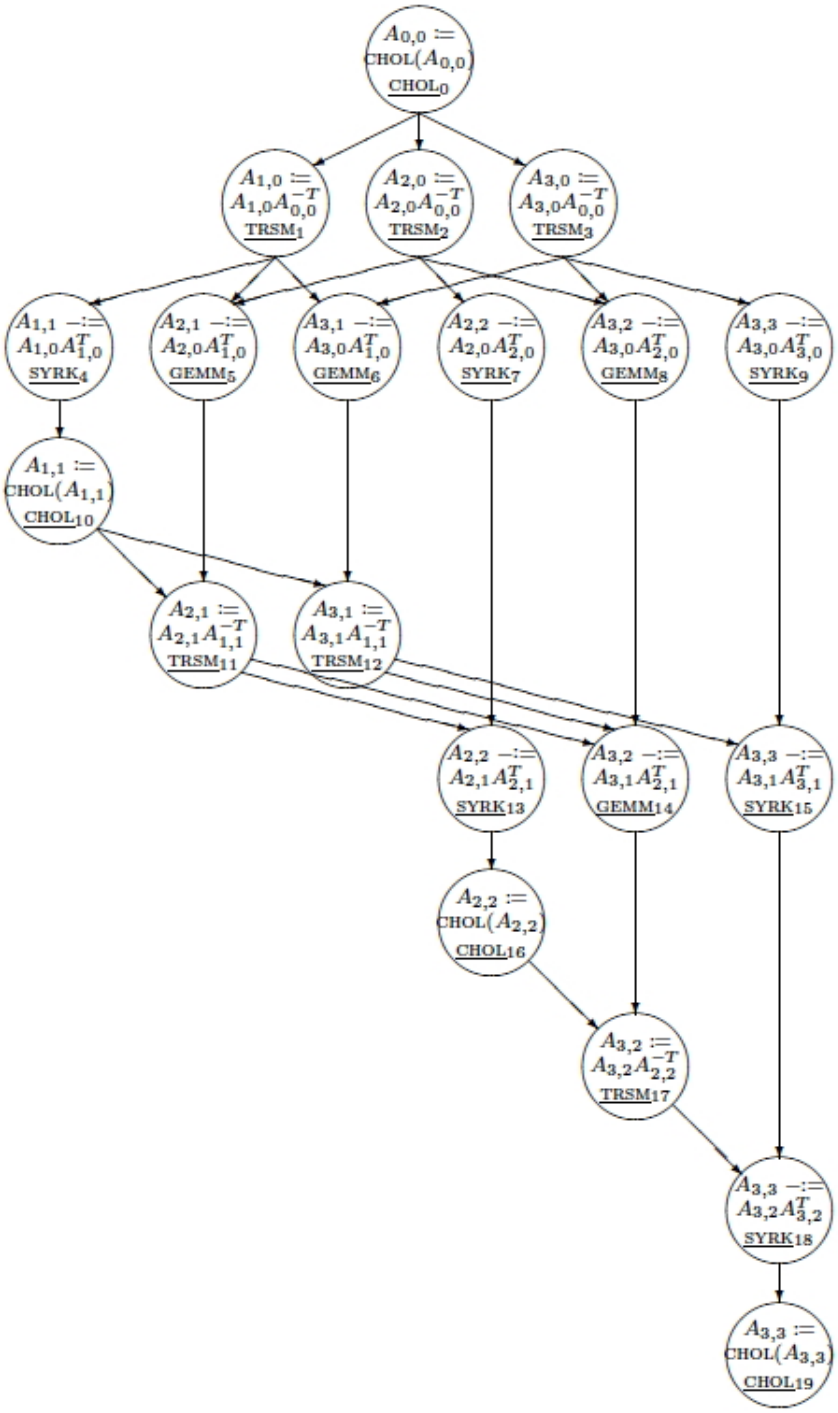


Figure 7.31: Graph of task dependencies in a 4×4 Cholesky factorization.

PART II

APPLICATIONS

Chapter 8

Molecular dynamics

Molecular dynamics is a technique for simulating the atom-by-atom behavior of molecules and deriving macroscopic properties from these atomistic motions. It has application to biological molecules such as proteins and nucleic acids, as well as natural and synthetic molecules in materials science and nanotechnology. Molecular dynamics falls in the category of particle methods, which includes N-body problems in celestial mechanics and astrophysics, and many of the ideas presented here will carry over to these other fields. In addition, there are special cases of molecular dynamics including ab initio molecular dynamics where electrons are treated quantum mechanically and thus chemical reactions can be modeled. We will not treat these special cases, but will instead concentrate on *classical* molecular dynamics.

The idea behind molecular dynamics is very simple: a set of particles interact according to Newton's law of motion, $F = ma$. Given the initial particle positions and velocities, the particle masses and other parameters, as well as a model of the forces that act between particles, Newton's law of motion can be integrated numerically to give a trajectory for each of the particles for all future (and past) time. Commonly, the particles reside in a computational box with periodic boundary conditions.

A molecular dynamics time step is thus composed of two parts:

1: compute forces on all particles
2: update positions (integration).

The computation of the forces is the expensive part. State-of-the-art molecular dynamics simulations are performed on parallel computers because the force computation is costly and a vast number of time steps are required for reasonable simulation lengths. In many cases, molecular dynamics is applied to simulations on molecules with a very large number of atoms as well, e.g., up to a million for biological molecules and long time scales, and up to billions for other molecules and shorter time scales.

Numerical integration techniques are also of interest in molecular dynamics. For simulations that take a large number of time steps and for which the preservation of quantities such as energy is more important than order of accuracy, the solvers that must be used are different than the traditional ODE solvers presented in Chapter 4.

In the following, we will introduce force fields used for biomolecular simulations and discuss fast methods for computing these forces. Then we devote sections to the parallelization of molecular dynamics for short-range forces and the parallelization of the 3-D FFT used in fast computations of long-range forces.

We end with a section introducing the class of integration techniques that are suitable for molecular dynamics simulations. Our treatment of the subject of molecular dynamics in this chapter is meant to be introductory and practical; for more information, the text [66] is recommended.

8.1 Force Computation

8.1.1 Force Fields

In classical molecular dynamics, the model of potential energy and of the forces that act between atoms is called a *force field*. The force field is a tractable but approximate model of quantum mechanical effects which are computationally too expensive to determine for large molecules. Different force fields are used for different types of molecules, as well as for the same molecule by different researchers, and none are ideal.

In biochemical systems, commonly-used force fields model the potential energy function as the sum of bonded, van der Waals, and electrostatic (Coulomb) energy:

$$E = E_{\text{bonded}} + E_{\text{Coul}} + E_{\text{vdW}}.$$

The potential is a function of the positions of all the atoms in the simulation. The force on an atom is the negative gradient of this potential at the position of the atom.

The bonded energy is due to covalent bonds in a molecule,

$$E_{\text{bonded}} = \sum_{\text{bonds}} k_i(r_i - r_{i,0})^2 + \sum_{\text{angles}} k_i(\theta_i - \theta_{i,0})^2 + \sum_{\text{torsions}} V_n(1 + \cos(n\omega - \gamma))$$

where the three terms are, respectively, sums over all covalent bonds, sums over all angles formed by two bonds, and sums over all dihedral angles formed by three bonds. The fixed parameters k_i, $r_{i,0}$, etc. depend on the types of atoms involved, and may differ for different force fields. Additional terms or terms with different functional forms are also commonly used.

The remaining two terms for the potential energy E are collectively called the nonbonded terms. Computationally, they form the bulk of the force calculation. The electrostatic energy is due to atomic charges and is modeled by the familiar

$$E_{\text{Coul}} = \sum_i \sum_{j>i} \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}}$$

where the sum is over all pairs of atoms, q_i and q_j are the charges on atoms i and j, and r_{ij} is the distance between atoms i and j. Finally, the van der Waals energy approximates the remaining attractive and repulsive effects, and is commonly modeled by the Lennard-Jones function

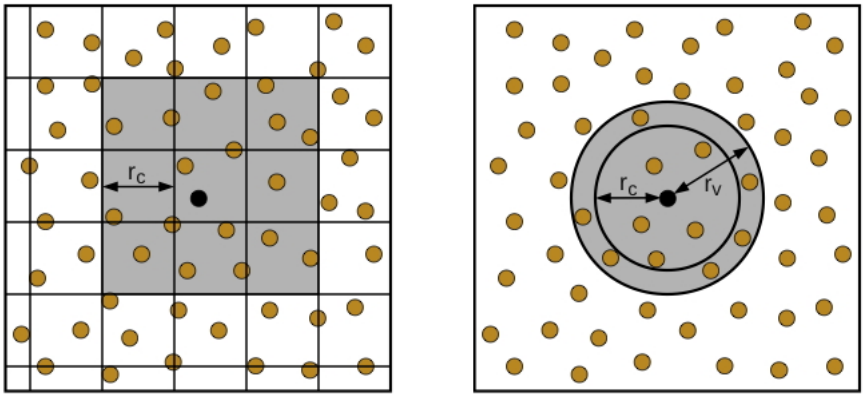
$$E_{\text{vdW}} = \sum_i \sum_{j>i} 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right]$$

where ϵ_{ij} and σ_{ij} are force field parameters depending on atom types. At short distances, the repulsive (r^{12}) term is in effect, while at long distances, the dispersive (attractive, $-r^6$) term is in effect.

Parallelization of the molecular dynamics force calculation depends on parallelization each of these individual types of force calculations. The bonded forces are local computations in the sense that for a given atom, only nearby atom positions and data are needed. The van der Waals forces are also local and are termed short-range because they are negligible for large atom separations. The electrostatic forces are long-range, and various techniques have been developed to speed up these calculations. In the next two subsections, we separately discuss the computation of short-range and long-range nonbonded forces.

8.1.2 Computing Short-Range Nonbonded Forces

The computation of short-range nonbonded forces for a particle can be truncated beyond a cutoff radius, r_c, of that particle. The naive approach to perform this computation for a particle i is by examining all other particles and computing their distance to particle i. For n particles, the complexity of this approach is $O(n^2)$, which is equivalent to computing forces between all pairs of particles. There are two data structures, *cell lists* and *Verlet neighbor lists*, that can be used independently for speeding up this calculation, as well as an approach that combines the two.



(a) Cell list method. (b) Verlet neighbor list method.

Figure 8.1: Computing nonbonded forces within a cutoff, r_c. To compute forces involving the highlighted particle, only particles in the shaded regions are considered.

8.1.2.1 Cell Lists

The idea of cell lists appears often in problems where a set of points that are nearby a given point is sought. Referring to Fig. 8.1(a), where we illustrate the idea with a 2-D example, a grid is laid over the set of particles. If the grid spacing is no less than r_c, then to compute the forces on particle i, only the particles in the cell containing i and the 8 adjacent cells need to be considered. One sweep through all the particles is used to construct a list of particles for each cell. These cell lists are used to compute the forces for all particles. At the next time step, since the particles have moved, the cell lists must be regenerated or updated. The complexity of this approach is $O(n)$ for computing the data structure and $O(n \times n_c)$ for

the force computation, where n_c is the average number of particles in 9 cells (27 cells in 3-D). The storage required for the cell list data structure is $O(n)$.

8.1.2.2 Verlet Neighbor Lists

The cell list structure is somewhat inefficient because, for each particle i, n_c particles are considered, but this is much more than the number of particles within the cutoff r_c. A Verlet neighbor list is a list of particles within the cutoff for a particle i. Each particle has its own list, and thus the storage required is $O(n \times n_v)$ where n_v is the average number of particles within the cutoff. Once these lists are constructed, computing the forces is then very fast, requiring the minimal complexity $O(n \times n_v)$. Constructing the list is more expensive, requiring examining all the particles for each particle, i.e., no less than the original complexity of $O(n^2)$. The advantage, however, is that the neighbor lists can be reused for many time steps if an expanded cutoff, r_v is used. Referring to a 2-D example in Fig. 8.1(b), the neighbor list can be reused as long as no particle from outside the two circles moves inside the inner circle. If the maximum speed of the particles can be estimated or bounded, then one can determine a number of time steps for which it is safe to reuse the neighbor lists. (Alternatively, it may be possible to signal when any particle crosses to a position within the cutoff.) Technically, the Verlet neighbor list is the list of particles within the expanded cutoff, r_v.

8.1.2.3 Using Cell and Neighbor Lists Together

The hybrid approach is simply to use Verlet neighbor lists but to use cell lists to construct the neighbor lists. This reduces the high cost when neighbor lists need to be regenerated. This hybrid approach is very effective and is often the approach used in state-of-the-art molecular dynamics software.

Both cell lists and Verlet neighbor lists can be modified to exploit the fact that the force f_{ij} on particle i due to particle j is equal to $-f_{ji}$ (Newton's third law) and only needs to be computed once. For example, for cell lists, only 4 of the 8 cells (in 2-D) need to be considered.

8.1.3 Computing Long-Range Forces

Electrostatic forces are challenging to compute because they are long-range: each particle feels a non-negligible electrostatic force from all other particles in the simulation. An approximation that is sometimes used is to truncate the force calculation for a particle after a certain cutoff radius (as is done for short-range van der Waals forces). This generally produces unacceptable artifacts in the results, however.

There are several more accurate methods for speeding up the computation of electrostatic forces, avoiding the $O(n^2)$ sum over all pairs of n particles. We briefly outline some of these methods here.

8.1.3.1 Hierarchical N-body Methods

Hierarchical N-body methods, including the Barnes-Hut method and the fast multipole method, are popular for astrophysical particle simulations, but are typically too costly for the accuracy required in biomolecular simulations. In the Barnes-Hut method, space is recursively divided into 8 equal cells (in 3-D) until each cell contains zero or one particles. Forces between nearby particles are computed individually, as normal, but for distant particles, forces are computed between one particle and a set of distant particles within a cell. An accuracy measure is used to determine if the force can be computed using a distant cell or

must be computed by individually considering its children cells. The Barnes-Hut method has complexity $O(n \log n)$. The fast multipole method has complexity $O(n)$; this method calculates the potential and does not calculate forces directly.

8.1.3.2 Particle-Mesh Methods

In particle-mesh methods, we exploit the Poisson equation

$$\nabla^2 \phi = -\frac{1}{\epsilon}\rho$$

which relates the potential ϕ to the charge density ρ, where $1/\epsilon$ is a constant of proportionality. To utilize this equation, we discretize space using a mesh, assign charges to the mesh points, solve Poisson's equation on the mesh to arrive at the potential on the mesh. The force is the negative gradient of the potential (for conservative forces such as electrostatic forces). A number of techniques have been developed for distributing point charges in space to a set of mesh points and also for numerically interpolating the force on the point charges due to the potentials at the mesh points. Many fast methods are available for solving the Poisson equation, including multigrid methods and fast Fourier transforms. With respect to terminology, particle-mesh methods are in contrast to the naive *particle-particle* method where forces are computed between all pairs of particles.

It turns out that particle-mesh methods are not very accurate, and a more accurate alternative is to split each force into a short-range, rapidly-varying part and a long-range, slowly-varying part:

$$f_{ij} = f_{ij}^{sr} + f_{ij}^{lr}.$$

One way to accomplish this is to weigh f by a function $h(r)$, which emphasizes the short-range part (small r) and by $1 - h(r)$ which emphasizes the long-range part (large r). The short-range part is computed by computing the interaction of all pairs of particles within a cutoff (a particle-particle method) and the long-range part is computed using the particle-mesh method. The resulting method, called particle-particle-particle-mesh (PPPM, or P^3M) is due to Hockney and Eastwood, in a series of papers beginning in 1973.

8.1.3.3 Ewald Method

The Ewald method is the most popular of the methods described so far for electrostatic forces in biomolecular simulations and was developed for the case of periodic boundary conditions. The structure of the method is similar to PPPM in that the force is split between short-range and long-range parts. Again, the short-range part is computed using particle-particle methods, and the long-range part is computed using Fourier transforms. Variants of the Ewald method are very similar to PPPM in that the long-range part uses a mesh, and fast Fourier transforms are used to solve the Poisson equation on the mesh. For additional details, see, for example [66]. In Section 8.3, we describe the parallelization of the 3-D FFT to solve the 3-D Poisson equation.

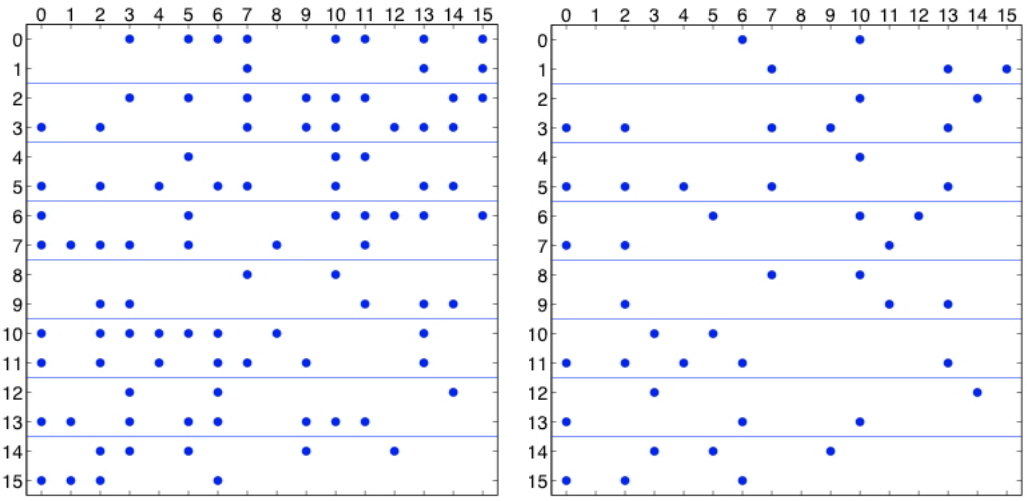
8.2 Parallel Decompositions

We now discuss the parallel computation of forces. Plimpton [163] created a very useful categorization of molecular dynamics parallelization methods, identifying *atom*, *force*, and *spatial* decomposition methods. Here, we closely follow his description of these methods. We also add a fourth category which has

come to be recognized as differing from the earlier categories, called *neutral territory* methods, a name coined by Shaw [175]. Neutral territory methods are currently used by many state-of-the-art molecular dynamics codes. Spatial decompositions and neutral territory methods are particularly advantageous for parallelizing cutoff-based calculations.

8.2.1 Atom Decompositions

In an atom decomposition, each particle is assigned to one processor, and that processor is responsible for computing the particle's forces and updating its position for the entire simulation. For the computation to be roughly balanced, each processor is assigned approximately the same number of particles (a random distribution works well). An important point of atom decompositions is that each processor generally needs to communicate with all other processors to share updated particle positions.



(a) Force matrix. (b) Force matrix, redundancies removed.

Figure 8.2: Atom decomposition, showing a force matrix of 16 particles distributed among 8 processors. A dot represents a nonzero entry in the force matrix. On the left, the matrix is symmetric; on the right, only one element of a pair of skew-symmetric elements is computed, to take advantage of Newton's third law.

An atom decomposition is illustrated by the *force matrix* in Fig. 8.2(a). For n particles, the force matrix is an n-by-n matrix; the rows and columns are numbered by particle indices. A nonzero entry f_{ij} in the matrix denotes a nonzero force on particle i due to particle j which must be computed. This force may be a nonbonded and/or a bonded force. When cutoffs are used, the matrix is sparse, as in this example. The matrix is dense if forces are computed between all pairs of particles. The matrix is skew-symmetric because of Newton's third law, $f_{ij} = -f_{ji}$. The lines in Fig. 8.2(a) show how the particles are partitioned. In the figure, 16 particles are partitioned among 8 processors.

Algorithm 1 shows one time step from the point of view of one processor. At the beginning of the time step, each processor holds the positions of particles assigned to it.

An optimization is to halve the amount of computation, which is possible because the force matrix is

Algorithm 1 Atom decomposition time step.

1: send/receive particle positions to/from all other processors
2: (if nonbonded cutoffs are used) determine which nonbonded forces need to be computed
3: compute forces for particles assigned to this processor
4: update positions (integration) for particles assigned to this processor

skew-symmetric. To do this, we choose exactly one of f_{ij} or f_{ji} for all skew-symmetric pairs such that each processor is responsible for computing approximately the same number of forces. Choosing the upper or lower triangular part of the force matrix is a bad choice because the computational load is unbalanced. A better choice is to compute f_{ij} if $i + j$ is even in the upper triangle, or if $i + j$ is odd in the lower triangle, as shown in Fig. 8.2(b). There are many other options.

When taking advantage of skew-symmetry in the force matrix, all the forces on a particle owned by a processor are no longer computed by that processor. For example, in Fig. 8.2(b), the forces on particle 1 are no longer computed only by the first processor. To complete the force calculation, processors must communicate to send forces that are needed by other processors and receive forces that are computed by other processors. The above algorithm must now be modified by adding a communication step (step 4) as shown in Algorithm 2.

Algorithm 2 Atom decomposition time step, without redundant calculations.

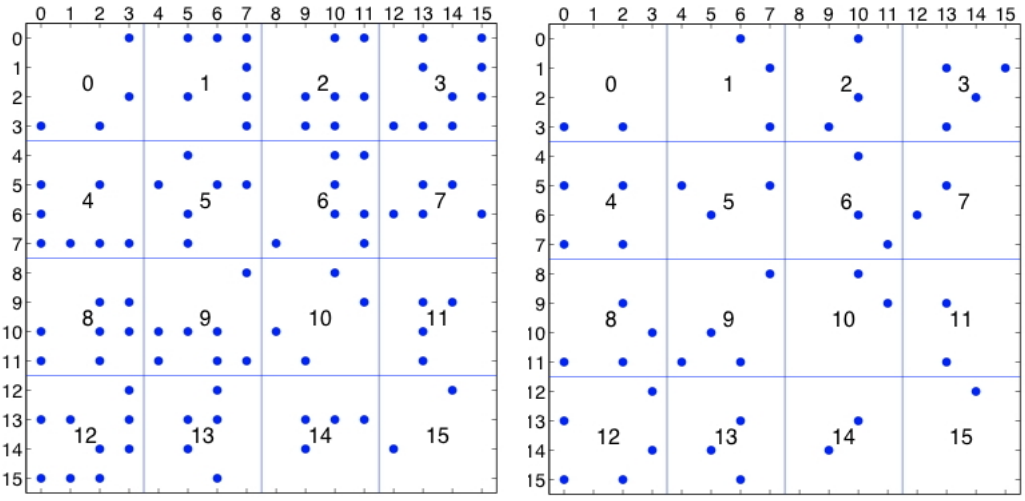
1: send/receive particle positions to/from all other processors
2: (if nonbonded cutoffs are used) determine which nonbonded forces need to be computed
3: compute *partial* forces for particles assigned to this processor
4: send particle forces needed by other processors and receive particle forces needed by this processor
5: update positions (integration) for particles assigned to this processor

This algorithm is advantageous if the extra communication is outweighed by the savings in computation. Note that the amount of communication doubles in general.

8.2.2 Force Decompositions

In a force decomposition, the forces are distributed among the processors for computation. A straightforward way to do this is to partition the force matrix into blocks and to assign each block to a processor. Fig. 8.3(a) illustrates this for the case of 16 particles and 16 processors. Particles also need to be assigned to processors (as in atom decompositions) for the purpose of having processors assigned to update particle positions. In the example of the Figure, processor i is assigned to update the positions of particle i; in practical problems, a processor would be assigned to update the positions of many particles. Note that, again, we first consider the case of a skew-symmetric force matrix.

We now examine the communication required in a time step for a force decomposition. Consider processor 3, which computes partial forces for particles 0, 1, 2, 3, and needs positions from particles 0, 1, 2, 3, and also 12, 13, 14, 15. Thus processor 3 needs to perform communication with processors 0, 1, 2, 3, and processors 12, 13, 14, 15. After forces have been computed by all processors, processor 3 needs to collect forces on



(a) Force matrix. (b) Force matrix, redundancies removed.

Figure 8.3: Force decomposition, showing a force matrix of 16 particles and forces partitioned among 16 processors.

particle 3 computed by other processors. Thus processor 2 needs to perform communication again with processors 0, 1, 2, 3.

Algorithm 3 shows what is performed in one time step, from the point-of-view of one processor. At the beginning of the time step, each processor holds the positions of all the particles assigned to it.

Algorithm 3 Force decomposition time step.

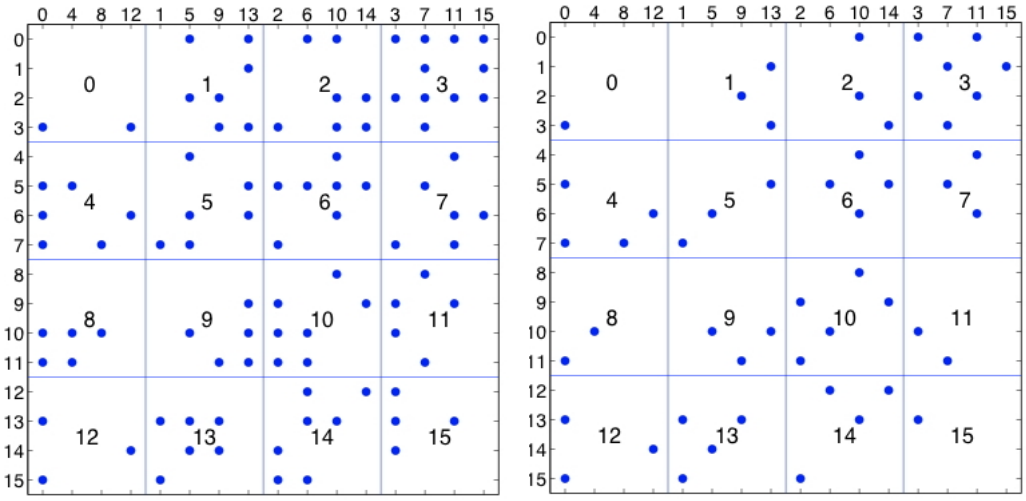
1: send positions of my assigned particles which are needed by other processors; receive *row* particle positions needed by my processor (this communication is between processors in the same processor row, e.g., processor 3 communicates with processors 0, 1, 2, 3)
2: receive *column* particle positions needed by my processor (this communication is generally with processors in another processor row, e.g., processor 3 communicates with processors 12, 13, 14, 15)
3: (if nonbonded cutoffs are used) determine which nonbonded forces need to be computed
4: compute forces for my assigned particles
5: send forces needed by other processors; receive forces needed for my assigned particles (this communication is between processors in the same processor row, e.g., processor 3 communicates with processors 0, 1, 2, 3)
6: update positions (integration) for my assigned particles

In general, if there are p processors (and p is square, for simplicity), then the the force matrix is partitioned into $\sqrt{p}$ by $\sqrt{p}$ blocks. The force decomposition just described requires a processor to communicate in three steps, with $\sqrt{p}$ processors in each step. This is much more efficient than atom decompositions which require communications among all p processors.

We can also exploit Newton's third law in force decompositions. Like for atom decompositions, we first

choose a modified force matrix where only one of f_{ij} and f_{ji} is computed. The forces on particle i are computed by a row of processors and now also by a column of processors. Thus an extra step of communication is needed by each processor to collect forces from a column of processors for particles assigned to it. Whereas there were three communication steps, there are now four communication steps when Newton's third law is exploited (the communication is not doubled in this case as in atom decompositions).

A modification to the force decomposition saves some communication. In Fig. 8.4, the columns are reordered using a *block-cyclic* ordering. Consider again processor 3, which computes partial forces for particles 0, 1, 2, 3. It needs positions from particles 0, 1, 2, 3, as before, but now also with processors 3, 7, 11, 15. The latter are processors in the same processor column as processor 3. Thus all communications are within the same processor row or processor column, which may be advantageous on mesh-based network architectures. The modified method is shown as Algorithm 4.



(a) Force matrix.

(b) Force matrix, redundancies removed.

Figure 8.4: Force decomposition, with permuted columns in the force matrix. Note that columns 3, 7, 11, 15 are now in the block column corresponding to processors 3, 7, 11, 15 (the same indices), etc.

8.2.3 Spatial Decompositions

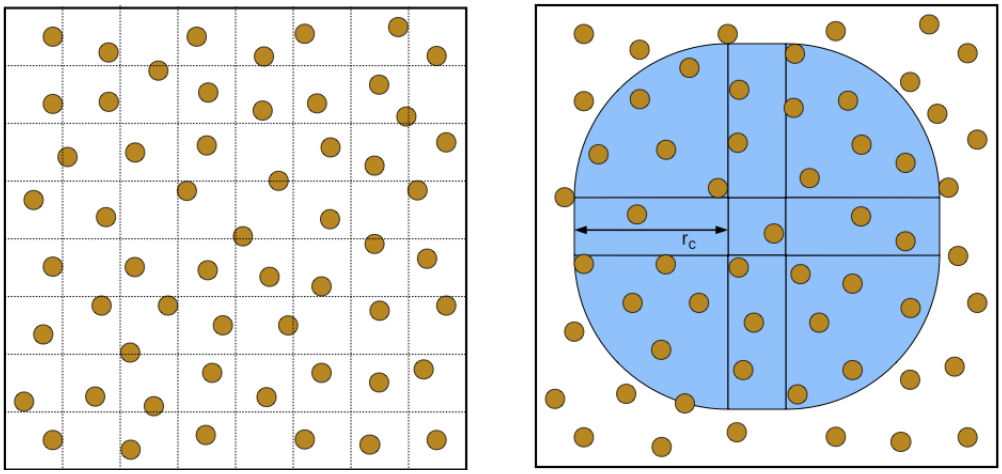
In a spatial decomposition, space is decomposed into cells. Each cell is assigned to a processor which is responsible for computing the forces on particles that lie inside the cell. Fig. 8.5(a) illustrates a spatial decomposition into 64 cells for the case of a 2-D simulation. (This is a decomposition of space and is not to be confused with a force matrix.) Typically, the number of cells is chosen to be equal to the number of processors. Since particles move during the simulation, the assignment of particles to cells changes as well. This is in contrast to atom and force decompositions.

Fig. 8.5(b) shows one cell (center square) and the region of space (shaded) that contains particles that are potentially within the cutoff radius, r_c, with particles in the given cell. The shaded region is often called the *import region*, since the given cell must import positions of particles lying in this region to perform its

Algorithm 4 Force decomposition time step, with permuted columns of force matrix.

1: send positions of my assigned particles which are needed by other processors; receive *row* particle positions needed by my processor (this communication is between processors in the same processor row, e.g., processor 3 communicates with processors 0, 1, 2, 3)
2: receive *column* particle positions needed by my processor (this communication is generally with processors the same processor column, e.g., processor 3 communicates with processors 3, 7, 11, 15)
3: (if nonbonded cutoffs are used) determine which nonbonded forces need to be computed
4: compute forces for my assigned particles
5: send forces needed by other processors; receive forces needed for my assigned particles (this communication is between processors in the same processor row, e.g., processor 3 communicates with processors 0, 1, 2, 3)
6: update positions (integration) for my assigned particles

force calculation. Note that not all particles in the given cell must interact with all particles in the import region, especially if the import region is large compared to the cutoff radius.



(a) Decomposition into 64 cells. (b) Import region for one cell.

Figure 8.5: Spatial decomposition, showing particles in a 2-D computational box, (a) partitioned into 64 cells, (b) import region for one cell.

Algorithm 5 shows what each processor performs in one time step. We assume that at the beginning of the time step, each processor holds the positions of the particles in its cell.

To exploit Newton's third law, the shape of the import region can be halved. Now each processor only computes a partial force on particles in its cell, and needs to receive forces from other processors to compute the total force on these particles. Thus an extra step of communication is involved. We leave it as an exercise to the reader to work out the details of the modified import region and the pseudocode for this case.

In the implementation of a spatial decomposition method, each cell is associated with a list of particles in

Algorithm 5 Spatial decomposition time step.

1: send positions needed by other processors for particles in their import regions; receive positions for particles in my import region
2: compute forces for my assigned particles
3: update positions (integration) for my assigned particles

its import region, similar to a Verlet neighbor list. Like a Verlet neighbor list, it is not necessary to update this list at every time step, if the import region is expanded slightly. This allows the import region list to be reused for several time steps, corresponding to the amount of time it takes a particle to traverse the width of the expanded region. This is exactly analogous to Verlet neighbor lists.

In summary, the main advantage of spatial decomposition methods is that they only require communication between processors corresponding to nearby particles. A disadvantage of spatial decomposition methods is that, for very large numbers of processors, the import region is large compared to the number of particles contained inside each cell.

8.2.4 Neutral Territory Methods

Our description of neutral territory methods follows closely that of Shaw [175]. A neutral territory method can be viewed as combining aspects of spatial decompositions and force decompositions. To parallelize the integration step, particles are assigned to processors according to a partitioning of space. To parallelize the force computation, each processor computes the forces between two sets of particles, but these particles may be unrelated to the particles that have been assigned to the processor for integration. As a result of this additional flexibility, neutral territory methods may require much less communication than spatial decomposition methods.

An example of a neutral territory method is shown in Fig. 8.6 for the case of a 2-D simulation. In the method shown in the Figure, the given processor is assigned the computation of forces between particles lying in the horizontal bar with particles lying in the vertical bar. These two regions thus form the import region for this method. By comparing to Fig. 8.6(b), the import region for this neutral territory method is much smaller than that for the corresponding spatial decomposition method. The advantage is greater when the size of the cells corresponding to each processor is small compared to the cutoff radius.

After the forces are computed, the given processor sends the forces it has computed to the processors that need these forces for integration. We thus have Algorithm 6.

Algorithm 6 Neutral territory method time step.

1: send and receive particle positions corresponding to import regions
2: compute forces assigned to this processor
3: send and receive forces required for integration
4: update positions (integration) for particles assigned to this processor

Like other methods, the import region of the neutral territory method can be modified to take advantage of Newton's third law. We refer to Shaw [175] for additional details and for illustrations of neutral territory

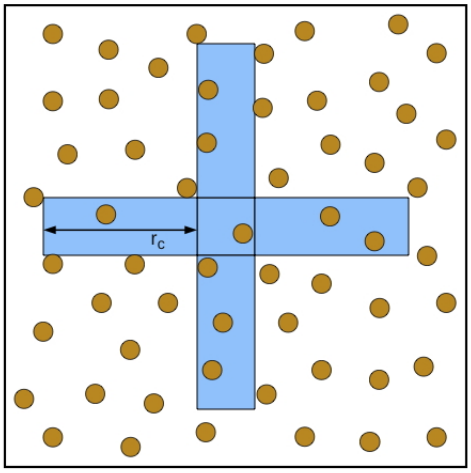


Figure 8.6: Neutral territory method, showing particles in a 2-D computational box and the import region (shaded) for one cell (center square). This Figure can be compared directly to the spatial decomposition case of Fig. 8.5(b). See Shaw [175] for additional details.

methods in 3-D simulations.

8.3 Parallel Fast Fourier Transform

A common component of many methods for computing long-range forces is the 3-D FFT for solving the Poisson equation on a 3-D mesh. The Fourier transform diagonalizes the Poisson operator (called the Laplacian) and one forward and one inverse FFT transformation are required in the solution. Consider the discrete Laplacian operator L (with periodic boundary conditions) and the solution of ϕ in $-L\phi = \rho$. Let F denote the Fourier transform. The original problem is equivalent to

$$\begin{aligned} -(FLF^{-1})F\phi &= F\rho \\ \phi &= -F^{-1}(FLF^{-1})^{-1}F\rho. \end{aligned}$$

The matrix FLF^{-1} is diagonal. The forward Fourier transform F is applied to ρ, then the Fourier-space components are scaled by the inverse of the diagonal matrix, and finally, the inverse Fourier transform F^{-1} is applied to obtain the solution ϕ.

For realistic protein sizes, a mesh spacing of approximately 1 Ångstrom is typically used, leading to a 3-D mesh that is quite small by many standards: $64 \times 64 \times 64$, or $128 \times 128 \times 128$. Parallel computation would often not be applied to a problem of this size, but parallel computation must be used because the data ρ is already distributed among the parallel processors (assuming a spatial decomposition is used).

A 3-D FFT is computed by computing 1-D FFTs in sequence along each of the three dimensions. For the $64 \times 64 \times 64$ mesh size, this is 4096 1-D FFTs of dimension 64. The parallel FFT calculation is typically bound by communication. The best parallelization of the FFT depends on the size of the transforms and the architecture of the computer network. Below, we first describe some concepts for parallel 1-D FFTs and then describe some concepts for parallel 3-D FFTs. For current software and research dedicated to the

parallelization and efficient computation (using SIMD operations) of large 1-D transforms, we refer to the SPIRAL and FFTW packages. These packages use autotuning to generate FFT codes that are efficient for the user's computer architecture.

8.3.1 Parallel 1-D FFT

8.3.1.1 1-D FFT without Transpose

Fig. 8.7 shows the data dependencies (data flow diagram) between the inputs (left) and outputs (right) for the 16-point radix-2 decimation-in-frequency FFT algorithm. (Not shown is a bit-reversal permutation that may be necessary in the computation.) The Figure also shows a partitioning of the computation among four processors. In this parallelization, the initial data is not moved among processors, but communication occurs during the computation. In the example shown in the Figure, communication occurs in the first two FFT stages; the final two stages do not involve communication. When communication does occur, every processor communicates with exactly one other processor.

8.3.1.2 1-D FFT with Transpose

Use of transposes is common to parallelize FFT computations. Fig. 8.8(a) shows the same data flow diagram as in Fig. 8.7, but horizontal lines have been removed and additional index labels have been added for clarity. As before, the first two FFT stages are performed without communication. The data is then transposed among the processors. With this transposed data layout, the last two FFT stages can be performed without communication. The final data is not in the original order; an additional transpose may be needed, or the data may be used in this transposed order. Fig. 8.8(b) shows how the indices are partitioned among four processors before and after the transpose. From these two Figures, notice that the first two stages have data dependencies that only involve indices in the same partition. The same is true for the second two stages for the partitioning after the transpose. Observe also that the structure of the computations before and after the transpose are identical.

8.3.2 Parallel 3-D FFT

8.3.2.1 3-D FFT with Block Decomposition

Fig. 8.9(a) shows a block decomposition of the FFT input data when a spatial decomposition is used for a mesh of size $8 \times 8 \times 8$ distributed across 64 processors arranged in a $4 \times 4 \times 4$ topology. The parallel 1-D FFT algorithms can be applied in each of the dimensions. For the example shown in the Figure, each 1-D FFT computation involves 4 processors. Each processor performs multiple 1-D FFTs simultaneously (four in this example). Within each processor, data is ordered contiguously if traversing one of the dimensions, and thus data access is strided for computation in the other two dimensions. Strided data access can be slow, and thus it may be worthwhile to reorder the data within each processor when computing the FFT for each of the dimensions.

8.3.2.2 3-D FFT with Slab Decomposition

The slab decomposition is shown in Fig. 8.9(b) for the case of 4 processors. Each processor holds one or more planes of the input data. This decomposition is used if the input data is already distributed in slabs,

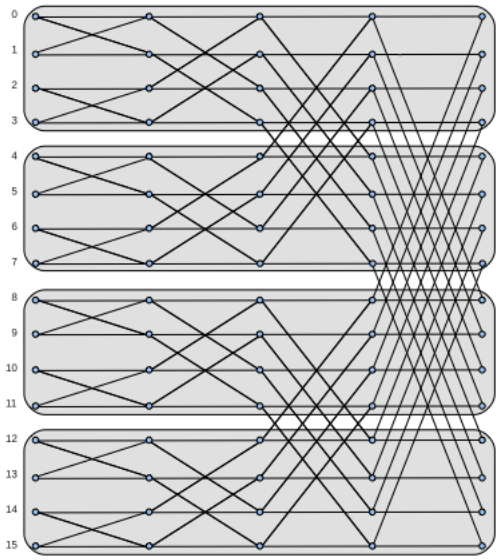


Figure 8.7: Data flow diagram for 1-D FFT for 16 points. The shaded regions show a decomposition for 4 processors (one processor per region). In this parallelization, the first two FFT stages have no communication; the last two FFT stages do have communication.

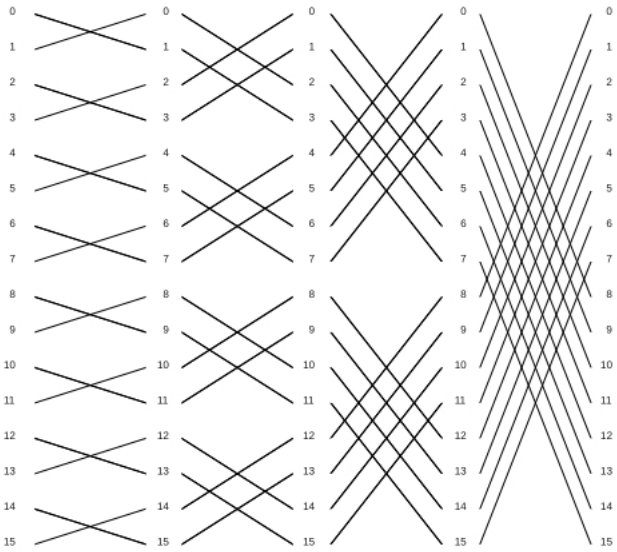
or if it can be redistributed this way. The two 1-D FFTs in the plane of the slabs require no communication. The remaining 1-D FFTs require communication and could use one of the two approaches for parallel 1-D FFTs described above. A disadvantage of the slab decomposition is that for large numbers of processors, the number of processors may exceed the number of points in the 3-D FFT along any one dimension. An alternative is the pencil decomposition below.

8.3.2.3 3-D FFT with Pencil Decomposition

The pencil decomposition is shown in Fig. 8.9(c) for the case of 16 processors. Each processor holds one or more pencils of the input data. If the original input data is distributed in blocks as in Fig. 8.9(a), then communication among a row of processors (in a 3-D processor mesh) can distribute the data into the pencil decomposition. The 1-D FFTs can then be performed with no communication. To perform the 1-D FFT in another dimension, the data needs to be redistributed into pencils in another dimension. In total, four communication stages are needed for the entire 3-D FFT computation.

8.4 Integration for Molecular Dynamics

To numerically integrate the system of ordinary differential equations in molecular dynamics, special methods are required, different than the traditional ODE solvers that were studied in Chapter 4. These special methods, called symplectic methods, are better than other methods at producing solutions that have constant energy, for example, for systems that are called Hamiltonian (which include systems from



(a) Data flow diagram (shown without horizontal lines for clarity) for 1-D FFT for 16 points.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

(b) Partitioning of the indices before (left) and after (right) the transpose.

Figure 8.8: 1-D FFT with transpose. The first two stages do not involve communication. The data is then transposed among the processors. As a result, the second two stages also do not involve communication.

molecular dynamics). When Hamiltonian systems are integrated with many time steps over a long time interval, preservation of structure such as total energy is often more important than the order of accuracy of the method. In this section, we motivate some ideas and give some details of the Störmer-Verlet method, which is sufficient for simple molecular dynamics simulations.

Hamiltonian systems are a class of dynamical systems which conserve energy and which can be written in a form called Hamilton's equations. Consider, for symplicity, the *simple harmonic oscillator*

$$u'' = -u$$

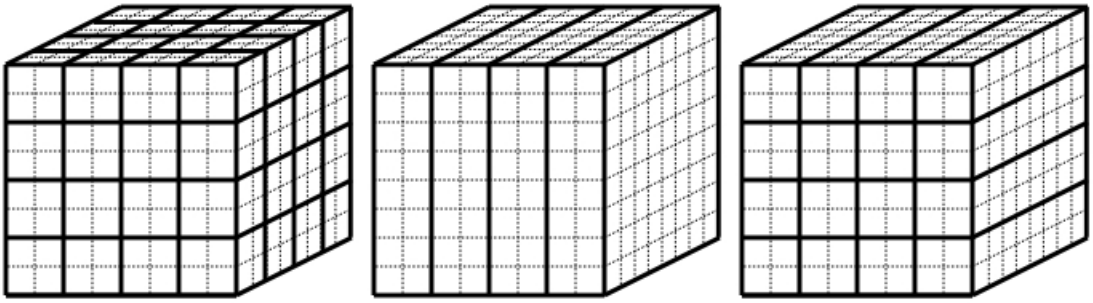
where u is the displacement of a single particle from an equilibrium point. This equation could model a particle with unit mass on a spring with unit spring constant. The force on a particle at position u is $-u$. This system does not look like a molecular dyanamics system but is useful for illustrating several ideas.

The above second order equation can be written as a system of first order equations

$$\begin{aligned} q' &= p \\ p' &= -q \end{aligned}$$

where $q = u$ and $p = u'$ which is common notation used in classical mechanics. The general solution is

$$\begin{pmatrix} q \\ p \end{pmatrix} = \begin{pmatrix} \cos t & \sin t \\ -\sin t & \cos t \end{pmatrix} \begin{pmatrix} q \\ p \end{pmatrix}.$$



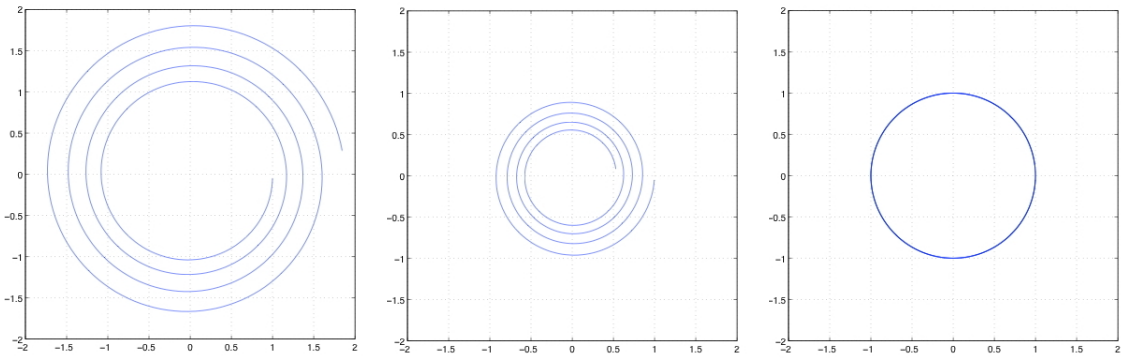
(a) Block Decomposition (b) Slab Decomposition (c) Pencil Decomposition

Figure 8.9: Three data decompositions for 3-D FFTs.

The kinetic energy of the simple harmonic oscillator is $p^2/2$ and the potential energy is $q^2/2$ (the negative gradient of potential energy is the force, $-q$). Thus the total energy is proportional to $q^2 + p^2$.

Now consider the solution of the system of first order equations by three methods, explicit Euler, implicit Euler, and a method called the Störmer-Verlet method. The initial condition is $(q, p) = (1, 0)$. We use a time step of $h = 0.05$ and take 500 steps. We plot q and p on the horizontal and vertical axes, respectively (called a *phase plot*). The exact solution, as given above, is a unit circle centered at the origin.

Figure 8.10 shows the solutions. For explicit Euler, the solution spirals outward, meaning the displacement and momentum of the solution increases over time. The opposite is true for the implicit Euler method. A plot of the total energy would show the energy increasing and decreasing for the two cases, respectively. The solutions are better when smaller time steps are taken or when higher order methods are used, but these methods are not at all appropriate for integration of symplectic systems over long periods of time. Figure 8.10(c) shows the solution using a symplectic method called the Störmer-Verlet method. The solution shows that $q^2 + p^2$ is preserved much better than in the other two methods.



(a) Explicit Euler (b) Implicit Euler (c) Störmer-Verlet Method

Figure 8.10: Phase plot of the solution of the simple harmonic oscillator for three methods with initial value (1,0), time step 0.05, and 500 steps. For explicit Euler, the solution spirals outward; for implicit Euler, the solution spirals inward; the total energy is best preserved with the Störmer-Verlet method.

We derive it the Störmer-Verlet method for the second order equation

$$u'' = f(t, u)$$

by simply replacing the left-hand side with a finite difference approximation

$$\frac{u_{k+1} - 2u_k + u_{k-1}}{h^2} = f(t_k, u_k)$$

which can be rearranged to obtain the method

$$u_{k+1} = 2u_k - u_{k-1} + h^2 f(t_k, u_k).$$

The formula can equivalently be derived from Taylor series. The method is similar to linear multistep methods in that some other technique is needed to supply the initial step of the method. The method is also time-reversible, because the formula is the same if $k + 1$ and $k - 1$ are swapped. To explain why this method is symplectic, unfortunately, is beyond the scope of this introduction.

The method as written above has a number of disadvantages, the most severe being that the addition of the small h^2 term is subject to catastrophic cancellation. Thus this formula should not be used in this form, and a number of mathematically equivalent formulas (which can be derived from the formula above) have been developed.

One alternative formula is the *leap-frog* method:

$$\begin{aligned} u_{k+1} &= u_k + hv_{k+1/2} \\ v_{k+1/2} &= v_{k-1/2} + hf(t_k, u_k) \end{aligned}$$

where v is the first derivative (velocity) which is offset from the displacement u by a half step. This formula does not suffer from the same roundoff problems and also makes available the velocities, although they need to be re-centered with displacements in order to calculate total energy at a given step. The second of this pair of equations is basically a finite difference formula.

A third form of the Störmer-Verlet method is the *velocity Verlet* variant:

$$\begin{aligned} u_{k+1} &= u_k + hv_k + \frac{h^2}{2} f(t_k, u_k) \\ v_{k+1} &= v_k + \frac{h^2}{2}(f(t_k, u_k) + f(t_{k+1}, u_{k+1})) \end{aligned}$$

where now the velocities are computed at the same points as the displacements. Each of these algorithms can be implemented such that only two sets of quantities need to be stored (two previous positions, or a position and a velocity). These variants of the Störmer-Verlet method are popular because of their simplicity, requiring only one costly force evaluation per step. Higher-order methods have generally not been practical.

The velocity Verlet scheme is also the basis of multiple time step algorithms for molecular dynamics. In these algorithms, the slowly-varying (typically long-range) forces are evaluated less frequently and update the positions less frequently than the quickly-varying (typically short-range) forces. Finally, many state-of-the-art molecular dynamics integrate a Hamiltonian system that has been modified in order to control the simulation temperature and pressure. Much more complicated symplectic methods have been developed for these systems.

Chapter 9

Combinatorial algorithms

In this chapter we will briefly consider a few combinatorial algorithms: sorting, and prime number finding with the Sieve of Eratosthenes.

Sorting is not a common operation in scientific computing: one expects it to be more important in databases, whether these be financial or biological (for instance in sequence alignment). However, it sometimes comes up, for instance in *Adaptive Mesh Refinement (AMR)* and other applications where significant manipulations of data structures occurs.

In this section we will briefly look at some basic algorithms and how they can be done in parallel. For more details, see [127] and the references therein.

9.1 Brief introduction to sorting

9.1.1 Complexity

There are many sorting algorithms. Traditionally, they have been distinguished by their computational complexity, that is, given an array of n elements, how many operations does it take to sort them, as a function of n.

Theoretically one can show that a sorting algorithm has to have at least complexity $O(n \log n)$. Consider a sorting algorithm as a decision tree: a first comparison is made, depending on it two other comparisons are made, et cetera. Thus, an actual sorting becomes a path through this decision tree. If every path has running time h, the tree has 2^h nodes. Since a sequence of n elements can be ordered in $n!$ ways, the tree needs to have enough paths to accomodate all of these; in other words, $2^h \geq n!$. Using Stirling's formula, this means that $n \geq O(n \log n)$.

There are indeed several algorithms that are guaranteed to attain this complexity, but a very popular algorithm, called *Quicksort* has only an 'expected' complexity of $O(n \log n)$, and a worst case complexity of $O(n^2)$. This behaviour results from the fact that quicksort needs to choose 'pivot elements' (we will go into more detail below in section 9.3), and if these choices are consistently the worst possible, the optimal complexity is not reached.

while *the input array has length* > 1 **do**
  Find a pivot element of intermediate size
  Split the array in two, based on the pivot
  Sort the two arrays.

Algorithm 2: The quicksort algorithm.

On the other hand, the very simple *bubble sort* algorithm always has the same complexity, since it has a static structure:

for *pass from* 1 *to* $n-1$ **do**
  for *e from 1 to* $n-$ *pass* **do**
    if *elements e and* $e+1$ *are ordered the wrong way* **then**
      exchange them

Algorithm 3: The bubble sort algorithm.

It is easy to see that this algorithm has a complexity of $O(n^2)$: the inner loop does t comparisons and up to t exchanges. Summing this from 1 to $n-1$ gives approximately $n^2/2$ comparisons and at most the same number of exchanges.

9.1.2 Sorting networks

Above we saw that some sorting algorithms operate independently of the actual input data, and some make decisions based on that data. The former class is sometimes called *sorting network*. It can be considered as custom hardware that implements just one algorithm. The basic hardware element is the *compare-and-swap* element, which has two inputs and two outputs. For two inputs x, y the outputs are $\max(x,y)$ and $\min(x,y)$.

In figure 9.1 we show buble sort, built up out of compare and swap elements.

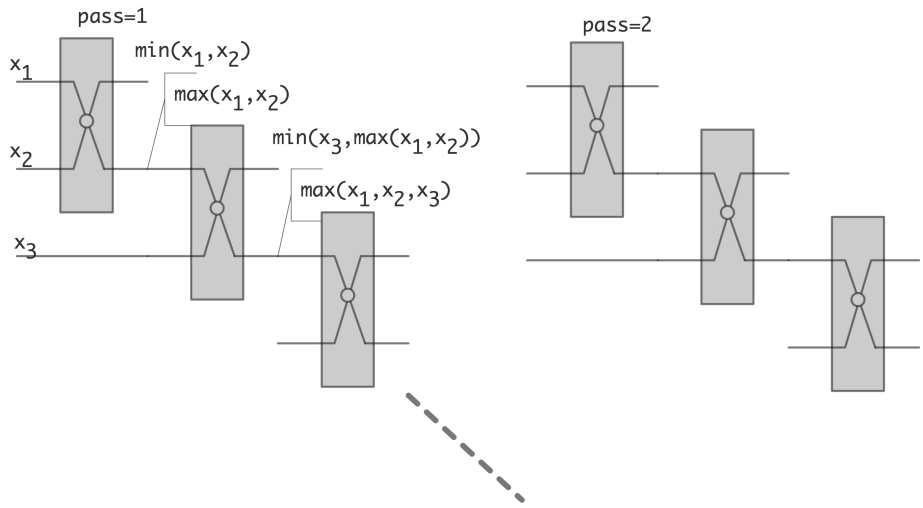


Figure 9.1: Bubble sort as a sorting network.

Below we will consider the Bitonic sort algorithm as a prime example of a sorting network.

9.1.3 Parallel and sequential complexity

Above we remarked that sorting sequentially takes at least $O(N \log N)$ time. If we can have perfect speedup, using for simplicity $P = N$ processors, we would have parallel time $O(\log N)$. If the parallel time is more than that, we define the *sequential complexity* as the total number of operations over all processors.

This equals the number of operations if the parallel algorithm were executed by one process, emulating all the others. Ideally this would be the same as the number of operations for a single-process algorithm, but it need not be. If it is larger, we have found another source of overhead: an intrinsic penalty for using a parallel algorithm. For instance, below we will see that sorting algorithm often have a sequential complexity of $O(N \log^2 N)$.

9.2 Odd-even transposition sort

Taking another look at figure 9.1, you see that the second pass can actually be started long before the first

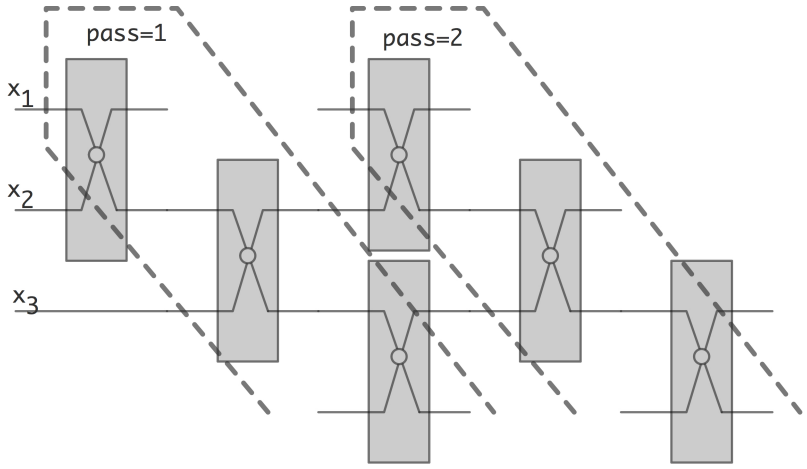


Figure 9.2: Overlapping passes in the bubble sort network.

pass is totally finished. This is illustrated in figure 9.2. If we now look at what happens at any given time, we obtain the *odd-even transposition sort* algorithm.

Odd-even transposition sort is a simple parallel sorting algorithm, with as main virtue that it is relatively easy to implement on a linear area of processors. On the other hand, it is not particularly efficient.

A single step of the algorithm consists of two substeps:

- Every even-numbered processor does a compare-and-swap with its right neighbor; then
- Every odd-numbered processor does a compare-and-swap with its right neighbor.

Theorem 3 *After $N/2$ steps, each consisting of the two substeps just given, a sequency is sorted.*

Proof. In each triplet $2i, 2i+1, 2i+2$, after an even and an odd step the largest element will be in rightmost position. Proceed by induction.

With a parallel time of N, this gives a sequential complexity N^2 compare-and-swap operations.

Exercise 9.1. Discuss speedup and efficiency of odd-even transposition sort sort, where we sort N numbers of P processors; for simplicity we set $N = P$ so that each processor contains a single number. Express execution time in *compare-and-swap* operations.

1. How many compare-and-swap operations does the parallel code take in total?
2. How many sequential steps does the algorithm take? What are T_1, T_p, T_∞, S_p, E_p for sorting N numbers? What is the average amount of parallelism?
3. Odd-even transposition sort sort can be considered a parallel implementation of bubble sort. Now let T_1 refer to the execution time of (sequential) bubble sort. How does this change S_p and E_p?

9.3 Quicksort

Quicksort is a recursive algorithm, that, unlike bubble sort, is not deterministic. It is a two step procedure, based on a reordering of the sequence[1]:

Algorithm: Dutch National Flag ordering of an array

Input : An array of elements, and a 'pivot' value

Output: The input array with elements ordered as red-white-blue, where red elements are larger than the pivot, white elements are equal to the pivot, and blue elements are less than the pivot

We state without proof that this can be done in $O(n)$ operations. With this, quicksort becomes:

Algorithm: Quicksort

Input : An array of elements

Output: The input array, sorted

while *The array is longer than one element* **do**

pick an arbitrary value as pivot

apply the Dutch National Flag reordering to this array

Quicksort(the blue elements)

Quicksort(the red elements)

The indeterminacy of this algorithm, and the variance in its complexity, stems from the pivot choice. In the worst case, the pivot is always the (unique) smallest element of the array. There will then be no blue elements, the only white element is the pivot, and the recursive call will be on the array of $n-1$ red elements. It is easy to see that the running time will then be $O(n^2)$. On the other hand, if the pivot is always (close to) the median, that is, the element that is intermediate in size, then the recursive calls will have an about equal running time, and we get a recursive formula for the running time:

$$T_n = 2T_{n/2} + O(n)$$

1. The name is explained by its origin with the Dutch computer scientist Edsger Dijkstra; see http://en.wikipedia.org/wiki/Dutch_national_flag_problem.

which is $O(n \log n)$; for proof, see section 15.2.

We will now consider parallel implementations of quicksort.

9.3.1 Quicksort in parallel

A simple parallelization of the quicksort algorithm can be achieved by executing the two recursive calls in parallel. This is easiest realized with a shared memory model, and threads (section 2.6.1) for the recursive calls.

```
function par_qs( data,nprocs ) {
  data_lo,data_hi = split(data);
  parallel do:
    par_qs( data_lo,nprocs/2 );
    par_qx( data_hi,nprocs/2 );
}
```

However, this implementation is not efficient without some effort. First of all, the recursion runs for $\log_2 n$ steps, and the amount of parallelism doubles in every step, so we could think that with n processing elements the whole algorithm takes $\log_2 n$ time. However, this ignores that each step does not take a certain unit time. The problem lies in the splitting step which is not trivially parallel.

Exercise 9.2. Make this argument precise. What is the total running time, the speedup, and the efficiency of parallelizing the quicksort algorithm this way?

Is there a way to make splitting the array more efficient? As it turns out, yes, and the key is to use a parallel *prefix operation*; see appendix 22. If the array of values is $x_1, \ldots, x_n$, we use a parallel prefix to compute how many elements are less than the pivot π:

$$X_i = \#\{x_j : j < i \wedge x_j < \pi\}.$$

With this, if a processor looks at x_i, and x_i is less than the pivot, it needs to be moved to location $X_i + 1$ in the array where elements are split according to the pivot. Similarly, one would count how many elements there are over the pivot, and move those accordingly.

This shows that each pivoting step can be done in $O(\log n)$ time, and since there $\log n$ steps to the sorting algorithm, the total algorithm runs in $O((\log n)^2)$ time. The *sequential complexity of quicksort* is $(\log_2 N)^2$.

9.3.2 Quicksort on a hypercube

As was apparent from the previous section, for an efficient parallelization of the quicksort algorithm, we need to make the Dutch National Flag reordering parallel too. Let us then assume that the array has been partitioned over the p processors of a hypercube of dimension d (meaning that $p = 2^d$).

In the first step of the parallel algorithm, we choose a pivot, and broadcast it to all processors. All processors will then apply the reordering independently on their local data.

In order to bring together the red and blue elements in this first level, every processor is now paired up with one that has a binary address that is the same in every bit but the most significant one. In each

pair, the blue elements are sent to the processor that has a 1 value in that bit; the red elements go to the processor that has a 0 value in that bit.

After this exchange (which is local, and therefore fully parallel), the processors with an address $1xxxxx$ have all the red elements, and the processors with an address $0xxxxx$ have all the blue elements. The previous steps can now be repeated on the subcubes.

This algorithm keeps all processors working in every step; however, it is susceptible to load imbalance if the chosen pivots are far from the median. Moreover, this load imbalance is not lessened during the sort process.

9.3.3 Quicksort on a general parallel processor

Quicksort can also be done on any parallel machine that has a linear ordering of the processors. We assume at first that every processor holds exactly one array element, and, because of the flag reordering, sorting will always involve a consecutive set of processors.

Parallel quicksort of an array (or subarray in a recursive call) starts by constructing a binary tree on the processors storing the array. A pivot value is chosen and broadcast through the tree. The tree structure is then used to count on each processor how many elements in the left and right subtree are less than, equal to, or more than the pivot value.

With this information, the root processor can compute where the red/white/blue regions are going to be stored. This information is sent down the tree, and every subtree computes the target locations for the elements in its subtree.

If we ignore network contention, the reordering can now be done in unit time, since each processor sends at most one element. This means that each stage only takes time in summing the number of blue and red elements in the subtrees, which is $O(\log n)$ on the top level, $O(\log n/2)$ on the next, et cetera. This makes for almost perfect speedup.

9.4 Radixsort

Most sorting algorithms are based on comparing the full item value. By contrast, *radix sort* does a number of partial sorting stages on the digits of the number. For each digit value a 'bin' is allocated, and numbers are moved into these bins. Concatenating these bins gives a partially sorted array, and by moving through the digit positions, the array gets increasingly sorted.

Consider an example with number of at most two digits, so two stages are needed:

array	25	52	71	12
last digit	5	2	1	2
(only bins 1,2,5 receive data)				
sorted	71	52	12	25
next digit	7	5	1	2
sorted	12	25	52	71

It is important that the partial ordering of one stage is preserved during the next. Inductively we then arrive at a totally sorted array in the end.

9.4.1 Parallel radix sort

A distributed memory sorting algorithm already has an obvious 'binning' of the data, so a natural parallel implementation of radix sort is based on using P, the number of processes, as radix.

We illustrate this with an example on two processors, meaning that we look at binary representations of the values.

	proc0		proc1	
array	2	5	7	1
binary	010	101	111	001
stage 1: sort by least significant bit				
last digit	0	1	1	1
	(this serves as bin number)			
sorted	010		101	111	001
stage 2: sort by middle bit				
next digit	1		0	1	0
	(this serves as bin number)			
sorted	101	001	010	111
stage 3: sort by most significant bit				
next digit	1	0	0	1
	(this serves as bin number)			
sorted	001	010	101	111
decimal	1	2	5	7

(We see that there can be load imbalance during the algorithm.)

Analysis:

- Determining the digits under consideration, and determining how many local values go into which bin are local operations. We can consider this as a connectivity matrix C where $C[i,j]$ is the amount of data that process i will send to process j. Each process owns its row of this matrix.
- In order to receive data in the shuffle, each process needs to know how much data will receive from every other process. This requires a 'transposition' of the connectivity matrix. In MPI terms, this is an *all-to-all* operation: `MPI_Alltoall`.
- After this, the actual data can be shuffled in another all-to-all operation. However, this the amounts differ per i,j combination, we need the `MPI_Alltoallv` routine.

9.4.2 Radix sort by most significant digit

It is perfectly possible to let the stages of the radix sort progress from most to least significant digit, rather than the reverse. Sequentially this does not change anything of significance.

However,

- Rather than full shuffles, we are shuffling in increasingly small subsets, so even the sequential algorithm will have increased *spatial locality*.
- A shared memory parallel version will show similar locality improvements.
- An MPI version no longer needs all-to-all operations. If we recognize that every next stage will be within a subset of processes, we can use the communicator splitting mechanism.

Exercise 9.3. The locality argument above was done somewhat hand-wavingly. Argue that the algorithm can be done in both a *breadth-first* and *depth-first* fashion. Discuss the relation of this distinction with the locality argument.

9.5 Samplesort

You saw in *Quicksort* (section 9.3) that it is possible to use probabilistic elements in a sorting algorithm. We can extend the idea of picking a single pivot, as in Quicksort, to that of picking as many pivots as there are processors. Instead of a *bisection* of the elements, this divides the elements into as many 'buckets' as there are processors. Each processor then sorts its elements fully in parallel.

Input : p: the number of processors, N: the numbers of elements to sort; $\{x_i\}_{i<N}$ the elements to sort
Let $x_0 = b_0 < b_1 < \cdots < b_{p-1} < b_p = x_N$ (where $x_N > x_{N-1}$ arbitrary)
for $i = 0, \ldots, p-1$ **do**
  Let $s_i = [b_i, \ldots b_{i+1} - 1]$
for $i = 0, \ldots, p-1$ **do**
  Assign the elements in s_i to processor i
for $i = 0, \ldots, p-1$ *in parallel* **do**
  Let processor i sort its elements

Algorithm 4: The Samplesort algorithm.

Clearly this algorithm can have severe *load imbalance* if the buckets are not chosen carefully. Randomly picking p elements is probably not good enough; instead, some form of *sampling* of the elements is needed. Correspondingly, this algorithm is known as **Samplesort** [17].

While the sorting of the buckets, once assigned, is fully parallel, this algorithm still has some problems regarding parallelism. First of all, the sampling is a sequential bottleneck for the algorithm. Also, the step where buckets are assigned to processors is essentially an *all-to-all* operation,

For an analysis of this, assume that there are P processes that first function as mappers, then as reducers. Let N be the number of data points, and define a block size $b \equiv N/P$. The cost of the processing steps of the algorithm is:

- The local determination of the bin for each element, which takes time $O(b)$; and

- The local sort, for which is we can assume an optimal complexity of $b \log b$.

However, the shuffle step is non-trivial. Unless the data is partially pre-sorted, we can expect the shuffle to be a full *all-to-all*, with a time complexity of $P\alpha + b\beta$. Also, this may become a network bottleneck. Note that in Quicksort on a hypercube there was never any *contention* for the wires.

Exercise 9.4. Argue that for a small number of processes, $P \ll N$, this algorithm has perfect speedup and a sequential complexity (see above) of $N \log N$.

Comparing this algorithm to sorting networks like bitonic sort this sorting algorithm looks considerable simpler: it has only a one-step network. The previous question argued that in the 'optimistic scaling' (work can increase while keeping number of processors constant) the sequential complexity is the same as for the sequential algorithm. However, in the weak scaling analysis where we increase work and processors proportionally, the sequential complexity is considerably worse.

Exercise 9.5. Consider the case where we scale both N, P, keeping b constant. Argue that in this case the shuffle step introduces an N^2 term into the algorithm.

9.5.1 Sorting through MapReduce

The *Terasort* benchmark concerns *sorting* on a large file-based dataset. Thus, it is somewhat of a standard in *big data* systems. In particular, *MapReduce* is a prime candidate; see http://perspectives.mvdirona.com/2008/07/hadoop-wins-terasort/.

Using MapReduce, the algorithm proceeds as follows:

- A set of key values is determined through sampling or from prior information: the keys are such that an approximately equal number of records is expected to fall in between each pair. The number of intervals equals the number of reducer processes.
- The mapper processes then produces key/value pairs where the key is the interval or reducer number.
- The reducer processes then perform a local sort.

We see that, modulo terminology changes, this is really samplesort.

9.6 Bitonic sort

To motivate bitonic sorting, suppose a sequence $x = \langle x_0, \ldots, x_n - 1 \rangle$ consists of an ascending followed by a descending part. Now split this sequence into two subsequences of equal length defined by:

$$\begin{aligned} s_1 &= \langle \min\{x_0, x_{n/2}\}, \ldots \min\{x_{n/2-1}, x_{n-1}\} \\ s_2 &= \langle \max\{x_0, x_{n/2}\}, \ldots \max\{x_{n/2-1}, x_{n-1}\} \end{aligned} \tag{9.1}$$

From the picture it is easy to see that s_1, s_2 are again sequences with an ascending and descending part. Moreover, all the elements in s_1 are less than all the elements in s_2.

We call (9.1) an ascending bitonic sorter, sinc the second subsequence contains elements larger than in the first. Likewise we can construct a descending sorter by reversing the roles of maximum and minimum.

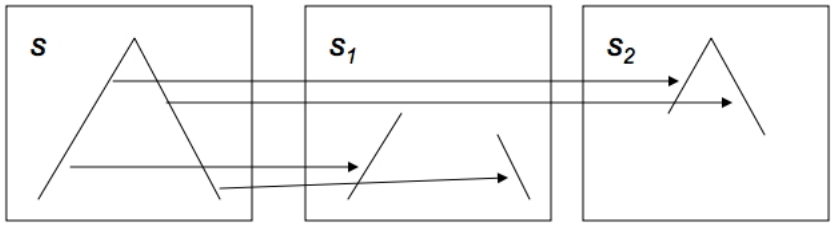


Figure 9.3: Illustration of a splitting a bitonic sequence.

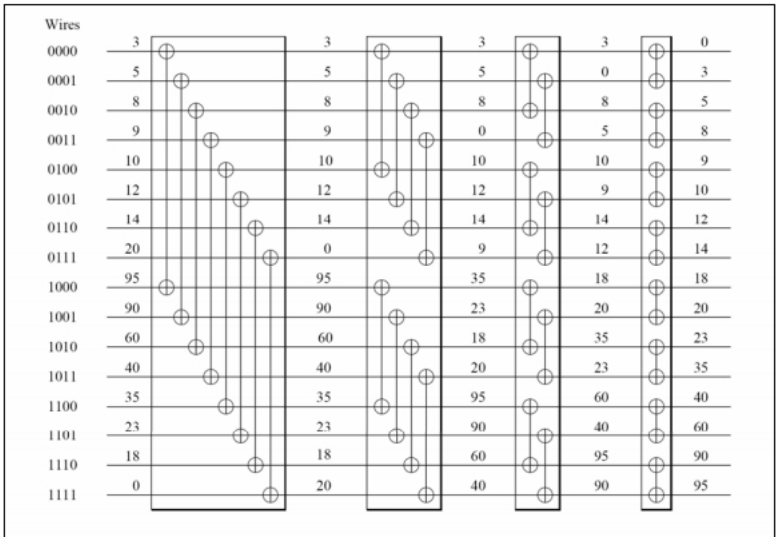


Figure 9.4: Illustration of a bitonic network that sorts a bitonic sequence of length 16.

It's not hard to imagine that this is a step in a sorting algorithm: starting out with a sequence on this form, recursive application of formula (9.1) gives a sorted sequence. Figure 9.4 shows how 4 bitonic sorters, over distances 8,4,2,1 respectively, will sort a sequence of length 16.

The actual definition of a *bitonic sequence* is slightly more complicated. A sequence is bitonic if it conists of an ascending part followed by a descending part, or is a cyclic permutation of such a sequence.

Exercise 9.6. Prove that splitting a bitonic sequence according to formula (9.1) gives two bitonic sequences.

So the question is how to get a bitonic sequence. The answer is to use larger and larger bitonic networks.

- A bitonic sort of two elements gives you a sorted sequence.
- If you have two sequences of length two, one sorted up, the other sorted down, that is a bitonic sequence.
- So this sequence of length four can be sorted in two bitonic steps.
- And two sorted sequences of length four form a bitonic sequence of length;
- which can be sorted in three bitonic steps; et cetera.

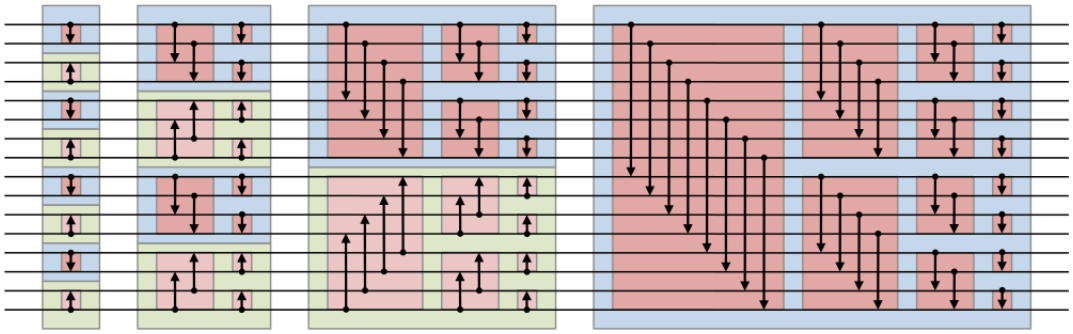


Figure 9.5: Full bitonic sort for 16 elements.

From this description you see that you $\log_2 N$ stages to sort N elements, where the i-th stage is of length $\log_2 i$. This makes the total *sequential complexity of bitonic sort* $(\log_2 N)^2$.

The sequence of operations in figure 9.5 is called a *sorting network*, built up out of simple *compare-and-swap* elements. There is no dependence on the value of the data elements, as is the case with quicksort.

9.7 Prime number finding

The *sieve of Eratosthenes* is a very old method for *finding prime numbers*. In a way, it is still the basis of many more modern methods.

Write down all natural numbers from 2 to some upper bound N, and we are going to mark these numbers as prime or definitely not-prime. All numbers are initially unmarked.

- The first unmarked number is 2: mark it as prime, and mark all its multiples as not-prime.
- The first unmarked number is 3: mark it as prime, and mark all its multiples as not-prime.
- The next number is 4, but it has been marked already, so mark 5 and its multiples.
- The next number is 6, but it has been marked already, so mark 7 and its multiples.
- Numbers 8,9,10 have been marked, so continue with 11.
- Et cetera.

Chapter 10

Graph analytics

Various problems in scientific computing can be formulated as graph problems; for an introduction to graph theory see Appendix 20). For instance, you have encountered the problems of load balancing (section 2.10.4) and finding independent sets (section 7.8.2).

Many traditional graph algorithms are not immediately, or at least not efficiently, applicable, since the graphs are often distributed, and traditional graph theory assumes global knowledge of the whole graph. Moreover, graph theory is often concerned with finding the optimal algorithm, which is usually not a parallel one. Conversely, in parallel we may get enough of a speedup that we can afford to waste some cycles on an algorithms that is not optimal sequentially. Therefore, parallel graph algorithms are a field of study by themselves.

Recently, new types of graph computations in have arisen in scientific computing. Here the graph are no longer tools, but objects of study themselves. Examples are the *World Wide Web* or the *social graph* of *Facebook*, or the graph of all possible *protein interactions* in a living organism.

For this reason, combinatorial computational science is becoming a discipline in its own right. In this section we look at *graph analytics*: computations on large graphs. We start by discussing some classic algorithms, but we give them in an algebraic framework that will make parallel implementation much easier.

10.1 Traditional graph algorithms

We start by looking at a few 'classic' graph algorithms, and we discuss how they can be implemented in parallel. The connection between graphs and sparse matrices (see Appendix 20) will be crucial here: many graph algorithms have the structure of a *sparse matrix-vector multiplication*.

10.1.1 Shortest path algorithms

There are several types of shortest path algorithms. For instance, in the *single source shortest path* algorithm one wants the shortest path from a given node to any other node. In *all-pairs shortest path* algorithms one wants to know the shortest distance between any two nodes. Computing the actual paths is not part

of these algorithms; however, it is usually easy to include some information by which the path can be reconstructed later.

We start with a simple algorithm: finding the single source shortest paths in an unweighted graph. This is simple to do with a Breadth-First Search (BFS): in each step we have a set U of nodes for which the shortest distances are known, and we consider the set V of their neighbors.

Input : A graph, and a starting node s
Output: A function $d(v)$ that measures the distance from s to v
Let s be given, and set $d(s) = 0$
Initialize the finished set as $U = \{s\}$
Set $c = 1$
while *not finished* **do**
  Let V be the set of neighbors of U that are not themselves in U
  if $V = \emptyset$ **then**
    We're done
  else
    for $v \in V$ **do**
      Set $d(v) = c + 1$
    $U \leftarrow U \cup V$
    Increase $c \leftarrow c + 1$

This way of formulating an algorithm is useful for theoretical purposes: typically you can formulate a *predicate* that is true for every iteration of the while loop. This then allows you to prove that the algorithm terminates and that it computes what you intended it to compute. And on a traditional processor this would indeed be how you program a graph algorithm. However, these days graphs such as from Facebook can be enormous, and you want to program your graph algorithm in parallel.

In that case, the traditional formulations fall short:

- They are often based on queues to which nodes are added or subtracted; this means that there is some form of shared memory.
- Statements about nodes and neighbors are made without wondering if these satisfy any sort of spatial locality; if a node is touched more than once there is no guarantee of temporal locality.

10.1.2 Floyd-Warshall all-pairs shortest path

The *Floyd-Warshall algorithm* is an example of an all-pairs shortest paths algorithm. It is a *dynamic programming* algorithm that is based on gradually increasing the set of intermediate nodes for the paths. Specifically, in step k all paths $u \rightsquigarrow v$ are considered that have intermediate nodes in the set $k = \{0, \ldots, k-1\}$, and $\Delta_k(u, v)$ is the defined as the path length from u to v where all intermediate nodes are in k. Initially, for $k = 0$ this means that there are no intermeidate edges, so only graph edges are considered; when $k \equiv |V|$ we have considered all possible paths and we are done.

The computational step is

$$\Delta_{k+1}(u, v) = \min\{\Delta_k(u, v), \Delta_k(u, k) + \Delta_k(k, v)\}. \tag{10.1}$$

that is, the k-th estimate for the distance $\Delta(u,v)$ is the minimum of the old one, and a new path that has become feasible now that we are considering node k. This latter path is found by concatenating paths $u \rightsquigarrow k$ and $k \rightsquigarrow v$.

Writing it algorithmically:

for *k from zero to* $|V|$ **do**
  for *all nodes* u, v **do**
    $\Delta_{uv} \leftarrow f(\Delta_{uv}, \Delta_{uk}, \Delta_{kv})$

we see that this algorithm has a similar structure to Gaussian elimination, except that there the inner loop would be 'for all $u, v > k$'. (One could also compare this to the *Gauss Jordan* algorithm.)

Algebraically:

for *k from zero to* $|V|$ **do**
  $D \leftarrow D._{\min}\big[D(:,k) \min \cdot_+ D(k,:)\big]$

The Floyd-Warshall algorithm does not tell you the actual path. Storing those paths during the distance calculation above is costly, both in time and memory. A simpler solution is possible: we store a second matrix $n(i,j)$ that has the highest node number of the path between i and j.

Exercise 10.1. Include the calculation of $n(i,j)$ in the Floyd-Warshall algorithm, and describe how to use this to find the shortest path between i and j once $d(\cdot,\cdot)$ and $n(\cdot,\cdot)$ are known.

In section 5.4.3 you saw that the factorization of sparse matrices leads to *fill-in*, so the same problem occurs here. This requires flexible data structures, and this problem becomes even worse in parallel; see section 10.5.

10.1.3 Spanning trees

In an undirected graph $G = \langle V, E\rangle$, we call $T \subset E$ a 'tree' if it is connected and acyclic. It is called a *spanning tree* if additionally its edges contain all vertices. If the graph has edge weights $w_i : i \in E$, the tree has weight $\sum_{e\in T} w_e$, and we call a tree a *minimum spanning tree* if it has minimum weight. A minimum spanning tree need not be unique.

Prim's algorithm, a slight variant of *Dijkstra's shortest path algorithm*, computes a spanning tree starting at a root. The root has path length zero, and all other nodes have path length infinity. In each step, all nodes connected to the known tree nodes are considered, and their best known path length updated.

Input : A graph $\langle V, E\rangle$ and node $s \in V$
Output: A function $\ell(\cdot)$ giving the shortest distance of any node to s
for *all vertices* v **do**
    $\ell(v) \leftarrow \infty$
$\ell(s) \leftarrow 0$
$Q \leftarrow V - \{s\}$ and $T \leftarrow \{s\}$
while $Q \neq \emptyset$ **do**
    let u be the element in Q with minimal $\ell(u)$ value
    remove u from Q, and add it to T
    for $v \in Q$ *with* $(u, v) \in E$ **do**
        if $\ell(u) + w_{uv} < \ell(v)$ **then**
            Set $\ell(v) \leftarrow \ell(u) + w_{uv}$

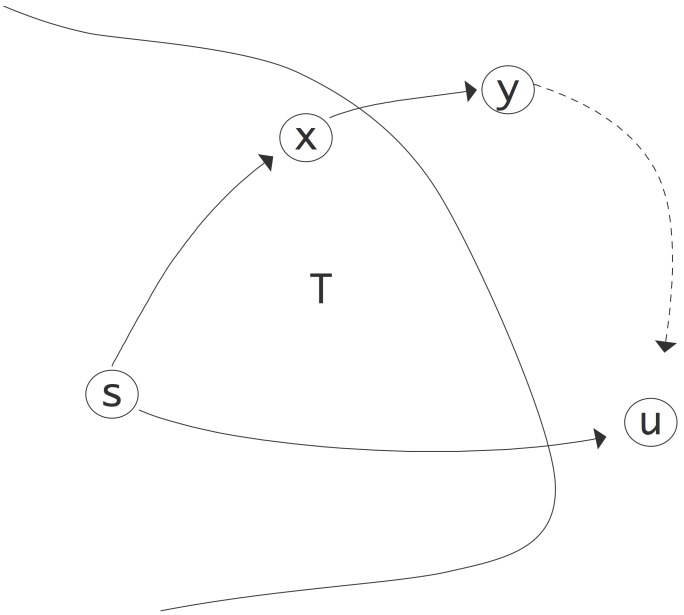


Figure 10.1: Illustration of the correctness of Dijkstra's algorithm.

Theorem 4 *The above algorithm computes the shortest distance from each node to the root node.*

Proof. The key to the correctness of this algorithm is the fact that we choose u to have minimal $\ell(u)$ value. Call the true shortest path length to a vertex $L(v)$. Since we start with an ℓ value of infinity and only ever decrease it, we always have $L(v) \leq \ell(v)$.

Our induction hypothesis will be that, at any stage in the algorithm, for the nodes in the current T the path length is determined correctly:

$$u \in T \Rightarrow L(u) = \ell(u).$$

This is certainly true when the tree consists only of the root s. Now we need to prove the induction step: if for all nodes in the current tree the path length is correct, then we will also have $L(u) = \ell(u)$.

Suppose this is not true, and there is another path that is shorter. This path needs to go through some node y that is not currently in T; this illustrated in figure 10.1. Let x be the node in T on the purported shortest path right before y. Now we have $\ell(u) > L(u)$ since we do not have the right pathlength yet, and $L(u) > L(x) + w_{xy}$ since there is at least one edge (which has positive weight) between y and u. But $x \in T$, so $L(x) = \ell(x)$ and $L(x) + w_{xy} = \ell(x) + w_{xy}$. Now we observe that when x was added to T its neighbors were updated, so $\ell(y)$ is $\ell_x + w_{xy}$ or less. Stringing together these inequalities we find

$$\ell(y) < \ell(x) + w_{xy} = L(x) + w_{xy} < L(u) < \ell(u)$$

which contradicts the fact that we choose u to have minimal ℓ value.

To parallelize this algorithm we observe that the inner loop has independent iterations and is therefore parallelizable. However, each iteration of the outer loop has a choice that minimizes a function value. Computing this choice is a reduction operator, and subsequently it needs to be broadcast. This strategy makes the sequential time equal to $d \log P$ where d is the depth of the spanning tree.

On a single processor, finding the minimum value in an array is naively an $O(N)$ operation, but through the use of a *priority queue* this can be reduced to $O(\log N)$. For the parallel version of the spanning tree algorithm the corresponding term is $O(\log(N/P))$, not counting the $O(\log P)$ cost of the reduction.

The *Bellman-Ford* algorithm is similar to Dijkstra's but can handle negative edge weights. It runs in $O(|V| \cdot |E|)$ time.

10.1.4 Graph cutting

Sometimes you may want to partition a graph, for instance for purposes of parallel processing. If this is done by partitioning the vertices, you are as-it-were cutting edges, which is why this is known as a *vertex cut* partitioning.

There are various criteria for what makes a good vertex cut. For instance, you want the cut parts to be of roughly equal size, in order to balance out parallel work. Since the vertices often correspond to communication, you also want the number of vertices on the cut (or their sum of weights in case of a weighted graph) to be small. The *graph Laplacian* (section 20.6.1) is a popular algorithm for this.

The graph Laplacian algorithm has the advantage over traditional graph partitioning algorithms [141] that it is easier parallelizable. Another popular computational method uses a multi-level approach:

1. Partition a coarsened version of the graph;
2. Gradually uncoarsen the graph, adapting the partitioning.

Another example of graph cutting is the case of a *bipartite graph*: a graph with two classes of nodes, and only edges from the one class to the other. Such a graph can for instance model a population and a set of properties: edges denote that a person has a certain interest.

10.1.5 Parallization of graph algorithms

Many graph algorithms, such as in section 10.1, are not trivial to parallelize. Here are some considerations.

First of all, unlike in many other algorithms, it is hard to target the outermost loop level, since this is often a 'while' loop, making a parallel stopping test necessary. On the other hand, typically there are

macro steps that are sequential, but in which a number of variables are considered independently. Thus there is indeed parallelism to be exploited.

The independent work in graph algorithms is of an interesting structure. While we can identify 'for all' loops, which are candidates for parallelization, these are different from what we have seen before.

- The traditional formulations often feature sets of variables that are gradually built up or depleted. This is implementable by using a shared data structures and a *task queue*, but this limits the implementation to some form of shared memory.
- Next, while in each iteration operations are independent, the dynamic set on which they operate means that assignment of data elements to processors is tricky. A fixed assignment may lead to much idle time, but dynamic assignment carries large overhead.
- With dynamic task assignment, the algorithm will have little spatial or temporal locality.

10.1.5.1 Parallelizing the all-pairs algorithms

In the single-source shortest path algorithm (section 10.1.1) we didn't have much choice but to parallelize by distributing the vector rather than the matrix. This type of distribution is possible here too, and it corresponds to a one-dimensional distribution of the $D(\cdot,\cdot)$ quantities.

Exercise 10.2. Sketch the parallel implementation of this variant of the algorithm. Show that each k-th iteration involves a broadcast with processor k as root.

However, this approach runs into the same scaling problems as the matrix-vector product using a one-dimensional distribution of the matrix; see section 7.2.3. Therefore we need to use a two-dimensional distribution.

Exercise 10.3. Do the scaling analysis. In a weak scaling scenario with constant memory, what is the asymptotic efficiency?

Exercise 10.4. Sketch the Floyd-Warshall algorithm using a two-dimensional distribution of the $D(\cdot,\cdot)$ quantities.

Exercise 10.5. The parallel Floyd-Warshall algorithm performs quite some operations on zeros, certainly in the early stages. Can you design an algorithm that avoids those?

10.2 Linear algebra on the adjacency matrix

A graph is often conveniently described through its *adjacency matrix*; see section 20.5. In this section we discuss algorithms that operate on this matrix as a linear algebra object.

In many cases we actually need the left product, that is, multiplying the adjacency matrix from the left by a row vector. Let for example e_i be the vector with a 1 in location i and zero elsewhere. Then $e_i^t G$ has a one in every j that is a neighbor of i in the adjacency graph of G.

More general, if we compute $x^tG = y^t$ for a unit basis vector x, we have

$$\begin{cases} x_i \neq 0 \\ x_j = 0 \quad j \neq i \end{cases} \wedge G_{ij} \neq 0 \Rightarrow y_j \neq 0.$$

Informally we can say that G makes a transition from state i to state j.

Another way of looking at this, is that if $x_i \neq 0$, then $y_j \neq 0$ for those j for which $(i,j) \in E$.

10.2.1 State transitions and Markov chains

Often we have vectors with only nonnegative elements, which case

$$\begin{cases} x_i \neq 0 \\ x_j \geq 0 \end{cases} \quad \text{all } j \quad \wedge\, G_{ij} \neq 0 \Rightarrow y_j \neq 0.$$

This left matrix-vector product has a simple application: *Markov chains*. Suppose we have a system (see for instance 21) that can be in any of n states, and the probabilities of being in these states sum up to 1. We can then use the adjacency matrix to model state transitions.

Let G by an adjacency matrix with the property that all elements are non-negative. For a Markov process, the sum of the probabilities of all state transitions, from a given state, is 1. We now interpret elements of the adjaceny matrix g_{ij} as the probability of going from state i to state j. We now have that the elements in each row sum to 1.

Let x be a probability vector, that is, x_i is the nonegative probability of being in state i, and the elements in x sum to 1, then $y^t = x^t G$ describes these probabilities, after one state transition.

Exercise 10.6. For a vector x to be a proper probability vector, its elements need to sum to 1. Show that, with a matrix as just described, the elements of y again sum to 1.

10.2.2 Computational form of graph algorithms

Let's introduce two abstract operators $\bigoplus, \otimes$ for graph operations. For instance, computing a shortest distance, can be expressed as the iterative application of a basic step:

> The distance to node y_i is the minimum of the already known distance, and the distance to a node x_j plus the value of the arc a_{ij}.

We write this step more abstractly as

$$y_i \bigoplus a_{ij} \otimes x_j$$

by analogy with the matrix-vector multiplication

$$y_i \leftarrow y_i + a_{ij} x_j$$

but with the definitions

$$y_i \bigoplus \xi_j \equiv y_i \leftarrow \min\{y_i, \xi_j\} \qquad \text{and} \qquad a_{ij} \otimes x_j \equiv \begin{cases} x_j + a_{ij} & a_{ij} \neq 0 \wedge x_j \neq \infty \\ \infty & a_{ij} = 0 \vee x_j = \infty \end{cases}$$

The shortest distance algorithm now becomes:

until all distances computed
    for nodes i
        for nodes j
            $y_i \bigoplus a_{ij} \otimes x_j$

You may recognize that this looks like a sequence of matrix-vector products, and indeed, with the ordinary $+, \times$ operators that's what this algorithm would compute. We will continue to explore the similarity between graph algorithm and linear algebra, though we point out that the above $\bigoplus, \otimes$ operators are not linear, or even associative.

This formalism will be used in several algorithms below; for other applications see [129].

10.2.2.1 Algorithm transformations

Analogous to transformations of the matrix-vector product, we now explore different variants of this graph algorithm that result from loop transformations.

We start by transforming the above algorithm by exchanging the i, j indices, giving the following:

for nodes j
    for nodes i
        $y_i \bigoplus a_{ij} \otimes x_j$

Next we make the outer loop more efficient by removing certain iterations. We do this by observing that x_j is invariant in the inner loop, and that the update in the inner loop does nothing for $x_j = \infty$, so we restrict the outer loop to indices where $x_j < \infty$. Likewise we restrict the inner loop, and apply a notation

$$j \to i \equiv a_{ij} \neq 0.$$

Together this gives:

for nodes $\{j : x_j < \infty\}$
    for nodes $\{i : j \to i\}$
        $y_i \bigoplus a_{ij} \otimes x_j$

In this we recognize the classic formulation of graph algorithms:

let the set of known distances $D = \{\text{rootnode}\}$
until $D = E$, that is, distances known for all edges
    for each $j \in D$:
        for each i connected to j:
            add i to D, with distance $d_j + e_{j \to i}$

10.2.2.2 The 'push model'

In a picture, we see that we can call the algorithm we have just derived a 'push model', since it repeatedly pushes the value of one node onto its neighbors. This is analogous to one form of the matrix-vector product, where the result vector is the sum of columns of the matrix.

While the algorithm is intuitive, as regards performance it is not great.

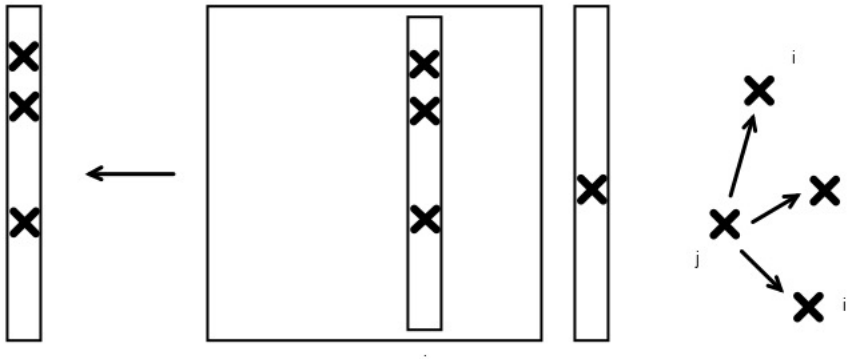


Figure 10.2: Push model of graph algorithms, depicted in matrix-vector form

- Since the result vector is repeatedly updated, it is hard to cache it.
- From nature of the push model, this variant is hard to parallelize, since it may involve conflicting updates. Thus a threaded code requires atomic operations, critical section, locks, or somesuch. A distributed memory code is even harder to write.

10.2.2.3 The 'pull' model

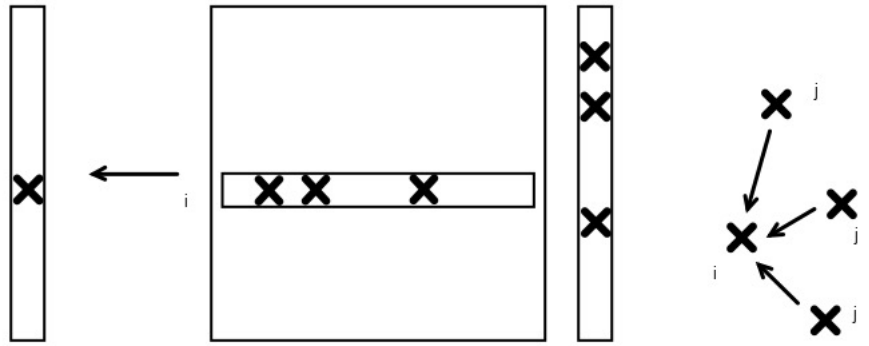


Figure 10.3: Pull model of graph algorithms, depicted in matrix-vector form

Leaving the original i, j nesting of the loops, we can not optimize the outer loop by removing certain iterations, so we wind up with:

for all nodes i
  for nodes $\{j : j \rightarrow i\}$
    $y_i \bigoplus a_{ij} \otimes x_j$

This is similar to the traditional inner-product based matrix vector product, but as a graph algorithm this variant is not often seen in the traditional literature. We can describe this as a 'pull model', since for each node i it pulls updates from all its neighbors j.

Regarding performance we can now state that:

- As a sequential algorithm, this can easily cache the output vector.
- This algorithm is easily parallizable over the i index; there are no conflicting writes.
- A distributed algorithm is also straightforward, with the computation preceeded by a (neighbor) allgather of the input vector.
- However, as a traditionally expressed graph algorithm it is wasteful, with many updates of each y_j component.

10.3 'Real world' graphs

In discussions such as in section 4.2.3 you have seen how the discretization of PDEs leads to computational problems that has a graph aspect to them. Such graphs have properties that make them amenable to certain kinds of problems. For instance, using FDMs or FEMs to model two or three-dimensional objects leads graphs where each node is connected to just a few neighbors. This makes it easy to find *separators*, which in turn allows such solution methods as *nested dissection*; see section 7.8.1.

There are however applications with computationally intensive graph problems that do not look like FEM graphs. We will briefly look at the example of the world-wide web, and algorithms such *Google*'s *PageRank* which try to find authoratative nodes.

For now, we will call such graphs *random graphs*, although this term has a technical meaning too [59].

10.3.1 Parallelization aspects

In section 7.5 we discussed the parallel evaluation of the sparse matrix-vector product. Because of the sparsity, only a partitioning by block rows or columns made sense. In effect, we let the partitioning be determined by one of the problem variables. This is also the only strategy that makes sense for single-source shortest path algorithms.

Exercise 10.7. Can you make an *a priori* argument for basing the parallelization on a distribution of the vector? How much data is involved in this operation, how much work, and how many sequential steps?

10.3.2 Properties of random graphs

The graphs we have seen in most of this course have properties that stem from the fact that they model objects in our three-dimensional world. Thus, the typical distance between two nodes is typically $O(N^{1/3})$ where N is the number of nodes. Random graphs do not behave like this: they often have a *small world* property where the typical distance is $O(\log N)$. A famous example is the graph of film actors and their connection by having appeared in the same movie: according to 'Six degrees of separation', no two actors have a distance more than six in this graph. In graph terms this means that the diameter of the graph is six.

Small-world graphs have other properties, such as the existence of cliques (although these feature too in higher order FEM problems) and hubs: nodes of a high degree. This leads to implications such as the following: deleting a random node in such a graph does not have a large effect on shortest paths.

Exercise 10.8. Considering the graph of airports and the routes that exist between them. If there are only hubs and non-hubs, argue that deleting a non-hub has no effect on shortest paths between other airports. On the other hand, consider the nodes ordered in a two-dimensional grid, and delete an arbitrary node. How many shortest paths are affected?

10.4 Hypertext algorithms

There are several algorithms based on linear algebra for measuring the importance of web sites [134] and ranking *web search* results. We will briefly define a few and discuss computational implications.

10.4.1 HITS

In the HITS (Hypertext-Induced Text Search) algorithm, sites have a *hub* score that measures how many other sites it points to, and an *authority* score that measures how many sites point to it. To calculate such scores we define an *incidence matrix* L, where

$$L_{ij} = \begin{cases} 1 & \text{document } i \text{ points to document } j \\ 0 & \text{otherwise} \end{cases}$$

The authority scores x_i are defined as the sum of the hub scores y_j of everything that points to i, and the other way around. Thus

$$\begin{aligned} x &= L^t y \\ y &= Lx \end{aligned}$$

or $x = LL^t x$ and $y = L^t L y$, showing that this is an eigenvalue problem. The eigenvector we need has only nonnegative entries; this is known as the *Perron vector* for a *nonnegative matrix*, see appendix 14.4. The Perron vector is computed by a *power method*; see section 14.3.

A practical search strategy is:

- Find all documents that contain the search terms;
- Build the subgraph of these documents, and possible one or two levels of documents related to them;
- Compute authority and hub scores on these documents, and present them to the user as an ordered list.

10.4.2 PageRank

The PageRank [159] basic idea is similar to HITS: it models the question 'if the user keeps clicking on links that are somehow the most desirable on a page, what will overall be the set of the most desirable links'. This is modeled by defining the ranking of a web page is the sum of the rank of all pages that connect to it. The algorithm It is often phrased iteratively:

while *Not converged* **do**
  for *all pages i* **do**
    $\text{rank}_i \leftarrow \epsilon + (1-\epsilon) \sum_{j\,:\,\text{connected} j \to i} \text{rank}_j$

  normalize the ranks vector

where the ranking is the fixed point of this algorithm. The ϵ term solve the problem that if a page has no outgoing links, a user that would wind up there would never leave.

Exercise 10.9. Argue that this algorithm can be interpreted in two different ways, roughly corresponding to the Jacobi and Gauss-Seidel iterative methods; section 5.5.3.

Exercise 10.10. In the PageRank algorithm, each page 'gives its rank' to the ones it connects to. Give pseudocode for this variant. Show that it corresponds to a matrix-vector product by columns, as opposed to by rows for the above formulation. What would be a problem implementing this in shared memory parallelism?

For an analysis of this method, including the question of whether it converges at all, it is better to couch it completely in linear algebra terms. Again we define a connectivity matrix

$$M_{ij} = \begin{cases} 1 & \text{if page } j \text{ links to } i \\ 0 & \text{otherwise} \end{cases}$$

With $e = (1, \ldots, 1)$, the vector $d^t = e^t M$ counts how many links there are on a page: d_i is the number of links on page i. We construct a diagonal matrix $D = \text{diag}(d_1, \ldots)$ we normalize M to $T = MD^{-1}$.

Now the columns sums (that is, the sum of the elements in any column) of T are all 1, which we can express as $e^t T = e^t$ where $e^t = (1, \ldots, 1)$. Such a matrix is called *stochastic matrix*. It has the following interpretation:

> If p is a *probability vector*, that is, p_i is the probability that the user is looking at page i, then Tp is the probability vector after the user has clicked on a random link.

Exercise 10.11. Mathematically, a probability vector is characterized by the fact that the sum of its elements is 1. Show that the product of a stochastic matrix and a probability vector is indeed again a probability vector.

The PageRank algorithm as formulated above would correspond to taking an arbitrary stochastic vector p, computing the *power method* $Tp, T^2p, T^3p, \ldots$ and seeing if that sequence converges to something.

There are few problems with this basic algorithm, such as pages with no outgoing links. In general, mathematically we are dealing with 'invariant subspaces'. Consider for instance an web with only 2 pages and the following *adjacency matrix*:

$$A = \begin{pmatrix} 1/2 & 0 \\ 1/2 & 1 \end{pmatrix}.$$

Check for yourself that this corresponds to the second page having no outgoing links. Now let p be the starting vector $p^t = (1, 1)$, and compute a few iterations of the power method. Do you see that the probability of the user being on the second page goes up to 1? The problem here is that we are dealing with a *reducible matrix*.

To prevent this problem, PageRank introduces another element: sometimes the user will get bored from clicking, and will go to an arbitrary page (there are also provisions for pages with no outgoing links). If we call s the chance that the user will click on a link, then the chance of going to an arbitrary page is $1-s$. Together, we now have the process

$$p' \leftarrow sTp + (1-s)e,$$

that is, if p is a vector of probabilities then p' is a vector of probabilities that describes where the user is after making one page transition, either by clicking on a link or by 'teleporting'.

The PageRank vector is the stationary point of this process; you can think of it as the probability distribution after the user has made infinitely many transitions. The PageRank vector satisfies

$$p = sTp + (1-s)e \Leftrightarrow (I - sT)p = (1-s)e.$$

Thus, we now have to wonder whether $I - sT$ has an inverse. If the inverse exists it satisfies

$$(I - sT)^{-1} = I + sT + s^2T^2 + \cdots$$

It is not hard to see that the inverse exists: with the Gershgorin theorem (appendix 14.5) you can see that the eigenvalues of T satisfy $|\lambda| \leq 1$. Now use that $s < 1$, so the series of partial sums converges.

The above formula for the inverse also indicates a way to compute the PageRank vector p by using a series of matrix-vector multiplications.

Exercise 10.12. Write pseudo-code for computing the PageRank vector, given the matrix T. Show that you never need to compute the powers of T explicitly. (This is an instance of *Horner's rule*).

In the case that $s = 1$, meaning that we rule out teleportation, the PageRank vector satisfies $p = Tp$, which is again the problem of finding the *Perron vector*; see appendix 14.4.

We find the Perron vector by a power iteration (section 14.3)

$$p^{(i+1)} = Tp^{(i)}.$$

This is a sparse matrix vector product, but unlike in the BVP case the sparsity is unlikely to have a structure such as bandedness. Computationally, one probably has to use the same parallelism arguments as for a dense matrix: the matrix has to be distributed two-dimensionally [154].

10.5 Large-scale computational graph theory

In the preceding sections you have seen that many graph algorithms have a computational structure that makes the matrix-vector product their most important kernel. Since most graphs are of low degree relative to the number of nodes, the product is a *sparse* matrix-vector product.

While the dense matrix-vector product relies on collectives (see section 7.2), the sparse case uses point-to-point communication, that is, every processor sends to just a few neighbors, and receives from just a

few. This makes sense for the type of sparse matrices that come from PDEs which have a clear structure, as you saw in section 4.2.3. However, there are sparse matrices that are so random that you essentially have to use dense techniques; see section 10.5.

In many cases we can then, as in section 7.5, make a one-dimensional distribution of the matrix, induced by a distribution of the graph nodes: if a processor owns a graph node i, it owns all the edges i, j.

However, often the computation is very unbalanced. For instance, in the single-source shortest path algorithm only the vertices along the front are active. For this reason, sometimes a distribution of edges rather than vertices makes sense. For an even balancing of the load even random distributions can be used.

The *parallelization of the Floyd-Warshall algorithm* (section 10.1.2) proceeds along different lines. Here we don't compute a quantity per node, but a quantity that is a function of pairs of nodes, that is, a matrix-like quantity. Thus, instead of distributing the nodes, we distribute the pair distances.

10.5.1 Distribution of nodes vs edges

Sometimes we need to go beyond a one-dimensional decomposition in dealing with sparse graph matrices. Let's assume that we're dealing with a graph that has a structure that is more or less random, for instance in the sense that the chance of there being an edge is the same for any pair of vertices. Also assuming that we have a large number of vertices and edges, every processor will then store a certain number of vertices. The conclusion is then that the chance that any pair of processors needs to exchange a message is the same, so the number of messages is $O(P)$. (Another way of visualizing this is to see that nonzeros are randomly distributed through the matrix.) This does not give a scalable algorithm.

The way out is to treat this sparse matrix as a dense one, and invoke the arguments from section 7.2.3 to decide on a two-dimensional distribution of the matrix. (See [201] for an application to the BFS problem; they formulate their algorithm in graph terms, but the structure of the 2D matrix-vector product is clearly recognizable.) The two-dimensional product algorithm only needs collectives in processor rows and columns, so the number of processors involved is $O(\sqrt{P})$.

10.5.2 Hyper-sparse data structures

With a two-dimensional distribution of sparse matrices, it is conceivable that some processes wind up with matrices that have empty rows. Formally, a matrix is called *hyper-sparse* if the number of nonzeros is (asymptotically) less than the dimension.

In this case, CRS format can be inefficient, and something called *double-compressed storage* can be used [24].

Chapter 11

N-body problems

In chapter 4 we looked at continuous phenomena, such as the behaviour of a heated rod in the entire interval $[0, 1]$ over a certain time period. There are also applications where you may be interested in a finite number of points. One such application is the study of collections of particles, possibly very big particles such as planets or stars, under the influence of a force such as gravity or the electrical force. (There can also be external forces, which we will ignore; also we assume there are no collisions, otherwise we need to incorporate nearest-neighbor interactions.) This type of problems is known as N-body problems; for an introduction see http://www.scholarpedia.org/article/N-body_simulations_(gravitational).

A basic algorithm for this problem is easy enough:

- choose some small time interval,
- calculate the forces on each particle, given the locations of all the particles,
- move the position of each particle as if the force on it stays constant throughout that interval.

For a small enough time interval this algorithm gives a reasonable approximation to the truth.

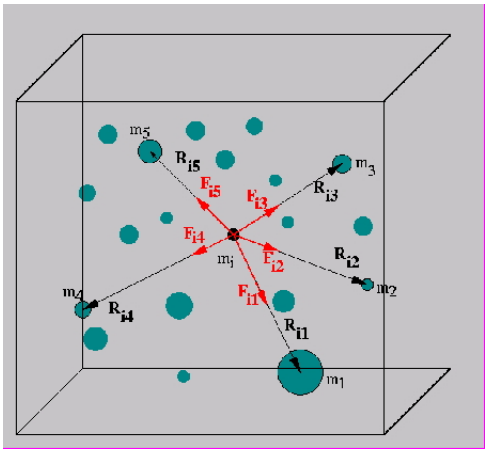


Figure 11.1: Summing all forces on a particle.

The last step, updating the particle positions, is easy and completely parallel: the problem is in evaluating the forces. In a naive way this calculation is simple enough, and even completely parallel:

for each particle i
    for each particle j
        let $\bar{r}_{ij}$ be the vector between i and j;
        then the force on i because of j is
        $f_{ij} = -\bar{r}_{ij} \frac{m_i m_j}{|r_{ij}|}$
        (where m_i, m_j are the masses or charges) and
        $f_{ji} = -f_{ij}$.

The main objection to this algorithm is that it has quadratic computational complexity: for N particles, the number of operations is $O(N^2)$.

Exercise 11.1. If we had N processors, the computations for one update step would take time $O(N)$. What is the communication complexity? Hint: is there a collective operations you can use?

Several algorithms have been invented to get the sequential complexity down to $O(N \log N)$ or even $O(N)$. As might be expected, these are harder to implement than the naive algorithm. We will discuss a popular method: the *Barnes-Hut algorithm* [7], which has $O(N \log N)$ complexity.

11.1 The Barnes-Hut algorithm

The basic observation that leads to a reduction in complexity is the following. If you are calculating the forces on two particles i_1, i_2 that are close together, coming from two particles j_1, j_2 that are also close together, you can clump j_1, j_2 together into one particle, and use that for both i_1, i_2.

Next, the algorithm uses a recursive division of space, in two dimensions in quadrants and in three dimensions in octants; see figure 11.2.

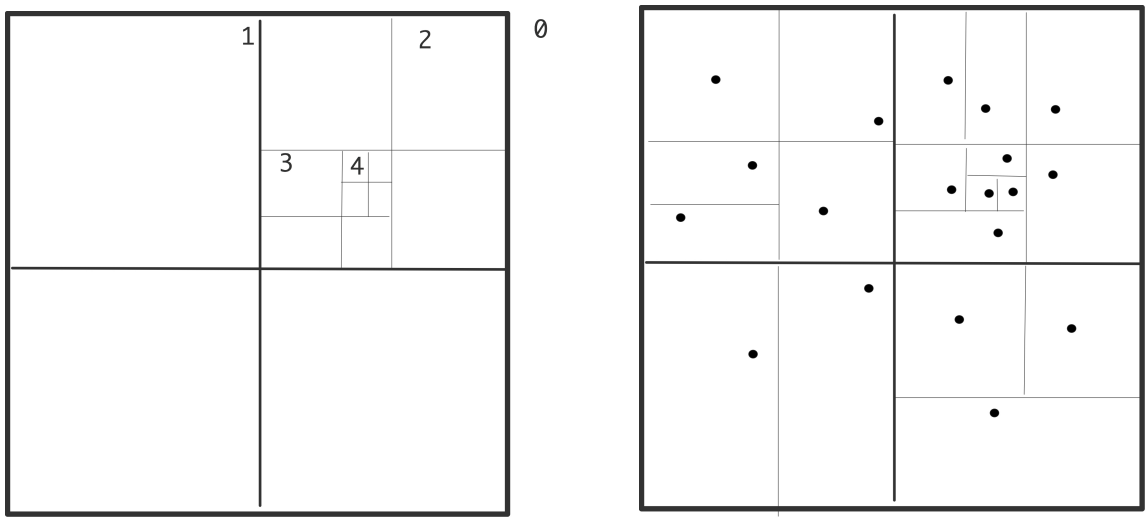


Figure 11.2: Recursive subdivision of a domain in quadrants with levels indicated (left); actual subdivision with one particle per box (right).

The algorithm is then as follows. First total mass and center of mass are computed for all cells on all levels:

```
for each level ℓ, from fine to coarse:
    for each cell c on level ℓ:
        compute the total mass and center of mass
                for cell c by considering its children
    if there are no particles in this cell,
        set its mass to zero
```

Then the levels are used to compute the interaction with each particle:

```
for each particle p:
    for each cell c on the top level
        if c is far enough away from p:
            use the total mass and center of mass of c;
        otherwise consider the children of c
```

The test on whether a cell is far enough away is typically implemented as the ratio of its diameter to its distance being small enough. This is sometimes referred to as the 'cell opening criterium'. In this manner, each particle interacts with a number of concentric rings of cells, each next ring of double width; see figure 11.3.

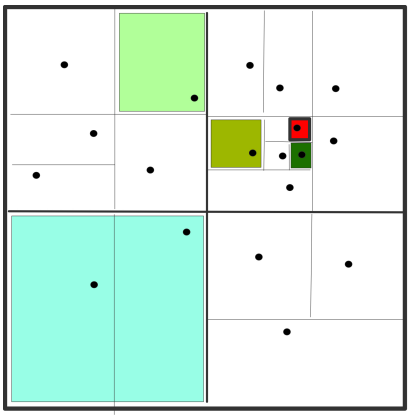


Figure 11.3: Boxes with constant distance/diameter ratio.

This algorithm is easy to realize if the cells are organized in a tree. In the three-dimensional case, each cell has eight children, so this is known as an *octtree*.

The computation of centres of masses has to be done each time after the particles move. Updating can be less expensive than computing from scratch. Also, it can happen that a particle crosses a cell border, in which case the data structure needs to be updated. In the worst case, a particle moves into a cell that used to be empty.

11.2 The Fast Multipole Method

The *Fast Multipole Method (FMM)* computes an expression for the potential at every point, not the force as does Barnes-Hut. FMM uses more information than the mass and center of the particles in a box. This more complicated expansion is more accurate, but also more expensive. In compensation, the FMM uses a fixed set of boxes to compute the potential, rather than a set varying with the accuracy parameter theta, and location of the center of mass.

However, computationally the FMM is much like the Barnes-Hut method so we will discuss their implementation jointly.

11.3 Full computation

Despite the above methods for judicious approximation, there are also efforts at full calculation of the N^2 interactions; see for instance the *NBODY6* code of Sverre Aarseth; see `http://www.ast.cam.ac.uk/~sverre/web/pages/home.htm`. Such codes use high order integrators and adaptive time steps. Fast implementation on the *Grape computer* exist; general parallelization is typically hard because of the need for regular load balancing.

11.4 Implementation

Octtree methods offer some challenges on high performance architectures. First of all, the problem is irregular, and secondly, the irregularity dynamically changes. The second aspect is mostly a problem in distributed memory, and it needs *load rebalancing*; see section 2.10. In this section we concentrated on the force calculation in a single step.

11.4.1 Vectorization

The structure of a problem as in figure 11.2 is quite irregular. This is a problem for vectorization on the small scale of *SSE/AVX* instructions and on the large scale of vector pipeline processors (see section 2.3.1 for an explanation of both). Program steps 'for all children of a certain box do something' will be of irregular length, and data will possibly be not stored in a regular manner.

This problem can be alleviated by subdividing the grid even if this means having empty boxes. If the bottom level is fully divided, there will always be eight (in three dimension) particles to operate on. Higher levels can also be filled in, but this means an increasing number of empty boxes on the lower levels, so there is a trade-off between increased work and increasing efficiency.

11.4.2 Shared memory implementation

Executed on a sequential architecture, this algorithm has complexity $O(N \log N)$. It is clear that this algorithm will also work on shared memory if each particle is turned into a task. Since not all cells contain particles, tasks will have a different running time.

11.4.3 Distributed memory implementation

The above shared-memory version of the Barnes-Hut algorithm can not immediately be used in a distributed memory context, since each particle can in principle access information from any part of the total data. It is possible to realize an implementation along these lines using a *hashed octtree*, but we will not persue this.

We observe data access is more structured than it seems at first. Consider a particle p and the cells on level ℓ that it interacts with. Particles located close to p will interact with the same cells, so we can rearrange interaction by looking at cells on level ℓ and the other cells on the same level that they interact with.

This gives us the following algorithm [120]: the calculation of centres of mass become a calculation of the force $g_p^{(\ell)}$ exerted *by* a particle p on level ℓ:

for level ℓ from one above the finest to the coarsest:
  for each cell c on level ℓ
    let $g_c^{(\ell)}$ be the combination of the $g_i^{(\ell+1)}$ for all children i of c

With this we compute the force *on* a cell:

for level ℓ from one below the coarses to the finest:
  for each cell c on level ℓ:

let $f_c^{(\ell)}$ be the sum of
1. the force $f_p^{(\ell-1)}$ on the parent p of c, and
2. the sums $g_i^{(\ell)}$ for all i on level ℓ that satisfy the cell opening criterium

We see that on each level, each cell now only interacts with a small number of neighbors on that level. In the first half of the algorithm we go up the tree using only parent-child relations between cells. Presumably this is fairly easy.

The second half of the algorithm uses more complicated data access. The cells i in the second term are all at some distance from the cell c on which we are computing the force. In graph terms these cells can be described as cousins: children of a sibling of c's parent. If the opening criterium is made sharper, we use second cousins: grandchildren of the sibling of c's grandparent, et cetera.

Exercise 11.2. Argue that this force calculation operation has much in common, structurally, with the sparse matrix-vector product.

In the shared memory case we already remarked that different subtrees take different time to process, but, since we are likely to have more tasks than processor cores, this will all even out. With distributed memory we lack the possibility to assign work to arbitrary processors, so we need to assign load carefully. Space-Filling Curves (SFCs) can be used here to good effect (see section 2.10.5.2).

11.4.4 1.5D implementation of the full method

It is possible to make a straightforward parallel implementation of the full N^2 method by distributing the particles, and let each particle evaluate the forces from every other particle.

Exercise 11.3. Since forces are symmetric, we can save a factor of two in work by letting particle p_i only interact with particles p_j with $j > i$. In this scheme, an equal distribution of data, e.g., particles, leads to an uneven distribution of work, e.g., interactions. How do the particles need to be distributed to get an even load distribution?

Assuming we don't care about the factor of two gain from using symmetry of forces, and using an even distribution of the particles, this scheme has a more serious problem in that it asymptotically does not scale.

The communication cost of this is

- $O(P)$ latency since messages need to be received from all processors.
- $O(N)$ bandwidth, since all particles need to be received.

If we however distribute the force calculations, rather than the particles, we come to different bounds; see [53] and references cited therein.

With every processor computing the interactions in a block with sides $N/\sqrt{P}$, there is replication of particles and forces need to be collected. Thus, the cost is now

- $O(\log p)$ latency for the broadcast and reduction
- $O(N/\sqrt{P} \cdot \log P)$ bandwidth, since that is how much force data each processor contributes to the final sum.

11.4.4.1 Problem description

We abstract the N-body problem as follows:

- We assume that there is a vector $c(\cdot)$ of the particle charges/masses and location information.
- To compute the location update, we need the pairwise interactions $F(\cdot,\cdot)$ based on storing tuples $C_{ij} = \langle c_i, c_j \rangle$.
- The interactions are then summed to a force vector $f(\cdot)$.

In the Integrative Model for Parallelism (IMP) model, the algorithm is sufficiently described if we know the respective data distributions; we are not immediately concerned with the local computations.

11.4.4.2 Particle distribution

An implementation based on particle distribution takes as its starting point the particle vector c distributed as $c(u)$, and directly computes the force vector $f(u)$ on the same distribution. We describe the computation as a sequence of three kernels, two of data movement and one of local computation[1]:

$$\begin{cases} \{\alpha : c(u)\} & \text{initial distribution for } c \text{ is } u \\ C(u,*) = c(u) \cdot_{\times} c(*) & \text{replicate } c \text{ on each processor} \\ \text{local computation of } F(u,*) \text{ from } C(u,*) & \\ f(u) = \sum_2 F(u,*) & \text{local reduction of partial forces} \\ \text{particle position update} & \end{cases} \tag{11.1}$$

The final kernel is a reduction with identical α and β-distribution, so it does not involve data motion. The local computation kernel also has not data motion. That leaves the gathering of $C(u,*)$ from the initial distribution $c(u)$. Absent any further information on the distribution u this is an allgather of N elements, at a cost of $\alpha \log p + \beta N$. In particular, the communication cost does not go down with p.

With a suitable programming system based on distributions, the system of equations (11.1) can be translated into code.

11.4.4.3 Work distribution

The particle distribution implementation used a one-dimensional distribution $F(u,*)$ for the forces conformally with the particle distribution. Since F is a two-dimensional object, it is also possible to use a two-dimensional distribution. We will now explore this option, which was earlier described in [53]; our purpose here is to show how this strategy can be implemented and analyzed in the IMP framework. Rather than expressing the algorithm as distributed computation and replicated data, we use a distributed temporary, and we derive the replication scheme as collectives on subcommunicators.

For a two-dimensional distribution of F we need $(N/b)\times(N/b)$ processors where b is a blocksize parameter. For ease of exposition we use $b = 1$, giving a number of processors $P = N^2$.

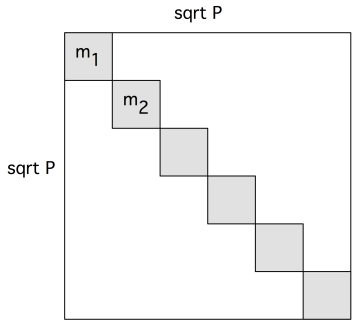
We will now go through the necessary reasoning. The full algorithm in close-to-implementable form is given in figure 11.4. For simplicity we use the identify distribution $I : p \mapsto \{p\}$.

1. In the full IMP theory [57] the first two kernels can be collapsed.

$$
\begin{array}{ll}
\{\alpha : C(D : \ < I, I >)\} & \text{initial distribution on the processor diagonal} \\
C(I,I) = C(I,p) + C(q,I) & \text{row and column broadcast of the processor diagonal} \\
F(I,I) \leftarrow C(I,I) & \text{local interaction calculation} \\
f(D : \ < I, I >) = \sum_2 F(I, D : *) & \text{summation from row replication on the diagonal}
\end{array}
\tag{11.2}
$$

Figure 11.4: IMP realization of the 1.5D N-body problem.

Initial store on the diagonal of the processor grid We first consider the gathering of the tuples $C_{ij} = \langle c_i, c_j \rangle$. Initially, we decide to let the c vector be stored on the diagonal of the processor grid; in other words, for all p, processor $\langle p, p \rangle$ contains C_{pp}, and the content of all other processors is undefined.



To express this formally, we let D be the diagonal $\{\langle p, p \rangle : \ p \in P\}$. Using the mechanism of partially defined distributions the initial α-distribution is then

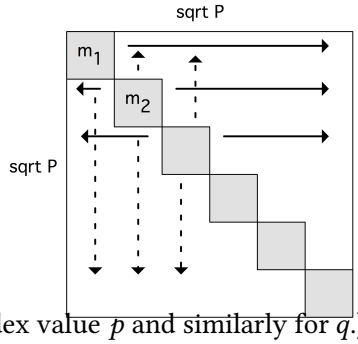
$$C(D : \ < I, I >) \equiv D \ni \langle p, q \rangle \mapsto C(I(p), I(q)) = C_{pq}. \tag{11.3}$$

Replication for force computation The local force computation $f_{pq} = f(C_{pq})$ needs for each processor $\langle p, q \rangle$ the quantity C_{pq}, so we need a β-distribution of $C(I, I)$. The key to our story is the realization that

$$C_{pq} = C_{pp} + C_{qq}$$

so that

$$C(I,I) = C(I,p) + C(q,I)$$



(where p stands for the distribution that maps each processor to the index value p and similarly for q.)

To find the transformation from α to β-distribution we consider the transformation of the expression $C(D : \ \langle I, I \rangle)$. In general, any set D can be written as a mapping from the first coordinate to sets of values of the second:

$$D \equiv p \mapsto D_p \qquad \text{where} \qquad D_p = \{q : \ \langle p, q \rangle \in D\}.$$

In our particular case of the diagonal we have

$$D \equiv p \mapsto \{p\}.$$

With this we write

$$C(D : \ \langle I, I \rangle) = C(I, D_p : \ I) = C(I, \{p\} : \ p). \tag{11.4}$$

Note that equation (11.4) is still the α-distribution. The β-distribution is $C(I, p)$, and it is an exercise in pattern matching to see that this is attained by a broadcast in each row, which carries a cost of

$$\alpha \log \sqrt{p} + \beta N/\sqrt{P}.$$

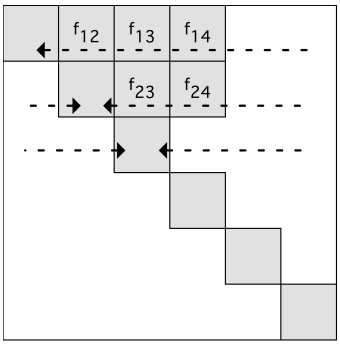
Likewise, $C(q, I)$ is found from the α-distribution by column broadcasts. We conclude that this variant does have a communication cost that goes down proportionally with the number of processors.

Local interaction calculation The calculation of $F(I, I) \leftarrow C(I, I)$ has identical α and β distributions, so it is trivialy parallel.

Summing of the forces We have to extend the force vector $f(\cdot)$ to our two-dimensional processor grid as $f(\cdot, \cdot)$. However, conforming to the initial particle distribution on the diagonal, we only sum on the diagonal. The instruction

$$f(D : \ < I, I >) = \sum_2 F(I, D : *)$$

has a β-distribution of $I, D : *$, which is formed from the α-distribution of I, I by gathering in the rows.



Chapter 12

Monte Carlo Methods

Monte Carlo simulation is a broad term for methods that use random numbers and statistical sampling to solve problems, rather than exact modeling. From the nature of this sampling, the result will have some uncertainty, but the statistical 'law of large numbers' will ensure that the uncertainty goes down as the number of samples grows.

An important tool for statistical sampling is a random number generator. See appendix 19 for random number generation.

12.1 Motivation

Let's start with a simple example: measuring an area, for instance, π is the area of a circle inscribed in a square with sides 2. If you picked a random point in the square, the chance of it falling in the circle is $\pi/4$, so you could estimate this ratio by taking many random points (x, y) and seeing in what proportion their length $\sqrt{x^2 + y^2}$ is less than 1.

You could even do this as a physical experiment: suppose you have a pond of an irregular shape in your backyard, and that the yard itself is rectangular with known dimensions. If you would now throw pebbles into your yard so that they are equally likely to land at any given spot, then the ratio of pebbles falling in the pond to those falling outside equals the ratio of the areas.

Less fanciful and more mathematically, we need to formalize the idea of falling inside or outside the shape you are measuring. Therefore, let $\Omega \in [0,1]^2$ be the shape, and let a function $f(\bar{x})$ describe the boundary of Ω, that is

$$\begin{cases} f(\bar{x}) < 0 & x \notin \Omega \\ f(\bar{x}) > 0 & x \in \Omega \end{cases}$$

Now take random points $\bar{x}_0, \bar{x}_1, \bar{x}_2 \in [0,1]^2$, then we can estimate the area of Ω by counting how often $f(\bar{x}_i)$ is positive or negative.

We can extend this idea to integration. The average value of a function on an interval (a, b) is defined as

$$\langle f \rangle = \frac{1}{b-a} \int_a^b f(x) dx$$

On the other hand, we can also estimate the average as

$$\langle f \rangle \approx \frac{1}{N} \sum_{i=1}^{n} f(x_i)$$

if the points x_i are reasonably distributed and the function f is not too wild. This leads us to

$$\int_a^b f(x)dx \approx (b-a)\frac{1}{N} \sum_{i=1}^{n} f(x_i)$$

Statistical theory, that we will not go into, tells us that the uncertainty σ_I in the integral is related to the standard deviation σ_f by

$$\sigma_I \sim \frac{1}{\sqrt{N}}\sigma_f$$

for normal distributions.

12.1.1 What is the attraction?

So far, Monte Carlo integration does not look much different from classical integration by *Riemann sums*. The difference appears when we go to higher dimensions. In that case, for classical integration we would need N points in each dimension, leading to N^d points in d dimensions. In the Monte Carlo method, on the other hand, the points are taken at random from the d-dimensional space, and a much lower number of points suffices.

Computationally, Monte Carlo methods are attractive since all function evaluations can be performed in parallel.

The statistical law that underlies this is as follows: if N independent observations are made of a quantity with standard deviation σ, then the standard deviation of the mean is $\sigma/\sqrt{N}$. This means that more observations will lead to more accuracy; what makes Monte Carlo methods interesting is that this gain in accuracy is not related to dimensionality of the original problem.

Monte Carlo techniques are of course natural candidatates for simulating phenomena that are statistical in nature, such as radioactive decay, or Brownian motion. Other problems where Monte Carlo simulation is attractive are outside the realm of scientific computing. For instance, the *Black-Scholes model* for stock *option pricing* [15] uses Monte Carlo simulation.

Some problems that you have seen before, such as solving a linear system of equations, can be tackled with Monte Carlo techniques. However, this is not a typical application. Below we will discuss two applications where exact methods would take far too much time to compute and where statistical sampling can quickly give a reasonably accurate answer.

12.2 Examples

12.2.1 Monte Carlo simulation of the Ising model

The Ising model (for an introduction, see [34]) was originally proposed to model ferromagnetism. Magnetism is the result of atoms aligning their 'spin' direction: let's say spin can only be 'up' or 'down', then a material has magnetism if more atoms have spin up than down, or the other way around. The atoms are said to be in a structure called a 'lattice'.

Now imagine heating up a material, which loosens up the atoms. If an external field is applied to the material, the atoms will start aligning with the field, and if the field is removed the magnetism disappears again. However, below a certain critical temperature the material will retain its magnetism. We will use Monte Carlo simulation to find the stable configurations that remain.

Let's say the lattice Λ has N atoms, and we denote a configuration of atoms as $\sigma = (\sigma_1, \ldots, \sigma_N)$ where each $\sigma_i = \pm 1$. The energy of a lattice is modeled as

$$H = H(\sigma) = -J \sum_i \sigma_i - E \sum_{ij} \sigma_i \sigma_j.$$

The first term models the interaction of individual spins σ_i with an external field of strength J. The second term, which sums over nearest neighbor pairs, models alignment of atom pairs: the product $\sigma_i \sigma_j$ is positive if the atoms have identical spin, and negative if opposite.

In *statistical mechanics*, the probability of a configuration is

$$P(\sigma) = \exp(-H(\sigma))/Z$$

where the 'partitioning function' Z is defined as

$$Z = \sum_\sigma \exp(H(\sigma))$$

where the sum runs over all 2^N configurations.

A configuration is stable if its energy does not decrease under small perturbations. To explore this, we iterate over the lattice, exploring whether altering the spin of atoms lowers the energy. We introduce an element of chance to prevent artificial solutions. (This is the *Metropolis algorithm* [149].)

for *fixed number of iterations* **do**
  for *each atom i* **do**
    calculate the change ΔE from changing the sign of σ_i
    if $\Delta E <$ *or* $\exp(-\Delta E)$ *greater than some random number* **then**
      accept the change

This algorithm can be parallelized, if we notice the similarity with the structure of the sparse matrix-vector product. In that algorithm too we compute a local quantity by combining inputs from a few nearest neighbors. This means we can partitioning the lattice, and compute the local updates after each processor collects a *ghost region*.

Having each processor iterate over local points in the lattice corresponds to a particular global ordering of the lattice; to make the parallel computation equivalent to a sequential one we also need a parallel random generator (section 19.3).

Chapter 13

Machine learning

Machine Learning (ML) is a collective name for a number of techniques that approach problems we might consider 'intelligent', such as image recognition. In the abstract, such problems are mappings from a vector space of *feature*s, such as pixel values in an image, to another vector space of outcomes. In the case of image recognition of letters, this final space could be 26-dimensional, and a maximum value in the second component would indicate that a 'B' was recognized.

The essential characteristic of ML techniques is that this mapping is described by a – usually large – number of internal parameters, and that these parameters are gradually refined. The learning aspect here is that refining the parameters happens by comparing an input to both its predicted output based on current parameters, and the intended output.

13.1 Neural networks

The most popular form of ML these days is Deep Learning (DL), or *neural networks*. A neural net, or a deep learning network, is a function that computes a numerical output, given a (typically multi-dimensional) input point. Assuming that this output is normalized to the interval $[0, 1]$, we can use a neural net as a classification tool by introducing a threshold on the output.

Why 'neural'?

A neuron, in a living body, is a cell that 'fires', that is, gives off a voltage spike, if it receives certain inputs. In ML we abstract this to a *perceptron*: a function that outputs a value depending on certain inputs. To be specific, the output value is often a linear function of the inputs, with the result limited to the range $(0, 1)$.

13.1.1 Single datapoint

In its simplest form we have an input, characterized by a vector of *feature s* $\bar{x}$, and a scalar output y. We can compute y as a linear function of $\bar{x}$ by using a vector of *weights* of the same size, and a scalar *bias* b:

$$y = \bar{w}\bar{x} + b.$$

13.1.2 Activation functions

To have some scale invariance, we introduce a function σ known as the *activation function* that maps $\mathbb{R} \to (0,1)$, and we actually compute the scalar output y as:

$$\mathbb{R} \ni y = \sigma(\bar{w}^t \bar{x} + b). \tag{13.1}$$

One popular choice for a *sigmoid* function is

$$\sigma(z) = \frac{1}{1+e^{-z}}.$$

This has the interesting property that

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

so computing both the function value and the derivative is not much more expensive than computing only the function value.

For vector-valued outputs we apply the sigmoid function in a pointwise manner:

MISSING SNIPPET netsigmoid in codesnippetsdir=snippets.code

Other activation functions are $y = \tanh(x)$ or 'ReLU' (Rectified Linear Unit)

$$f(x) = \max(0, x).$$

In other places (such as the final layer of a DL network) a *softmax* function may be more appropriate.

13.1.3 Multi-dimensional output

It is rare that a single layer, defined by $\bar{w}, \bar{b}$ can achieve all that we ask of a neural net. Typically we use the output of one layer as the input for a next layer. This means that instead of a scalar y we compute a multi-dimensional vector $\bar{y}$.

Now we have weights and a bias for each component of the output, so

$$\mathbb{R}^n \ni \bar{y} = \sigma(W\bar{x} + \bar{b})$$

where W is now a matrix.

A few observations:

- As indicated above, the output vector typically has fewer components than the input, so the matrix is not square, in particular not invertible.
- The sigmoid function makes the total mapping non-linear.
- Neural nets typically have multiple layers, each of which is a mapping of the form $x \to y$ as above.

13.1.4 Convolutions

The above discussion of applying weights considered the inputs as a set of features without further structure. However, in applications such as image recognition, where the input vector is an image, there is a structure to be acknowledged. Linearizing the input vector puts pixels close together in the input vector if they are close horizontally, but not vertically.

Thus we are motivated to find a weights matrix that reflects this locality. We do this by introducing *kernels*: a small 'stencil' that is applied at various points of the image. (See section 4.2.4 for a discussion of stencils in the context of PDEs.) Such a kernel is typically a small square matrix, and applying it is done by taking the inner product of the stencil values and the image values. (This is an inexact use of the term convolution from signal processing.)

Examples: https://aishack.in/tutorials/image-convolution-examples/.

13.2 Deep learning networks

We will now present a full neural network. This presentation follows [103].

Use a network with $L \geq 1$ layers, where layer $\ell = 1$ is the input layer, and layer $\ell = L$ is the output layer.

For $\ell = 1, \ldots, L$, layer ℓ compute

$$\begin{aligned} z^{(\ell)} &= W^{(\ell)} a^{(\ell)} + b^{(\ell)} \\ y^{(\ell)} &= \sigma\big(y^{(\ell)}\big) \end{aligned} \tag{13.2}$$

where $a^{(1)}$ is the input and $z^{(L+1)}$ is the final output.

We write this compactly as

$$y^{(L)} = N_{\{W^{(\ell)}\}_\ell, \{b^{(\ell)}\}_\ell ll}\big(a^{(1)}\big) \tag{13.3}$$

where we will usually omit the dependence of the net on the $W^{(\ell)}, b^{(\ell)}$ sets.

MISSING SNIPPET netforward in codesnippetsdir=snippets.code

MISSING SNIPPET layerforward in codesnippetsdir=snippets.code

13.2.1 Classification

In the above description both the input x and output y are vector-valued. There are also cases where a different type of output is desired. For instance, suppose we want to characterize bitmap images of digits; in that case the output should be an integer $0 \cdots 9$.

We accomodate this by letting the output y be in $\mathbb{R}^{10}$, and we say that the network recognizes the digit 5 if y_5 is sufficiently larger than the other output components. In this manner we keep the whole story still real-valued.

13.2.2 Error minimization

Often we have data points $\{x_i\}_{i=1,N}$ with known outputs y_i, and we want to make the network predict reproduce this mapping as well as possible. Formally, we seek to minimize the cost, or error:

$$C = \frac{1}{N} L(N(x_i), y_i)$$

over all choices $\{W\}, \{b\}$. (Usually we do not spell out explicitly that this cost is a function of all $W^{[\ell]}$ weight matrices and $b^{[\ell]}$ biases.)

MISSING SNIPPET netloss in codesnippetsdir=snippets.code

Minimizing the cost means to choose weights $\{W^{(\ell)}\}_\ell$ and biases $\{b^{(\ell)}\}_\ell$ such that for each x:

$$\left[\{W^{(\ell)}\}_\ell, \{b^{(\ell)}\}_\ell\right] = \underset{\{W\},\{b\}}{\operatorname{argmin}} L(N_{\{W\},\{b\}}(x), y) \tag{13.4}$$

where $L(N(x), y)$ is a *loss function* describing the distance between the computed output $N(x)$ and the intended output y.

We find this minimum using *gradient descent*:

$$w \leftarrow w + \Delta w, \qquad b \leftarrow b + \Delta b$$

where

$$\Delta W = \frac{\partial L}{\partial W_{ij}^{(\ell)}}$$

which is a complicated expression that we will now give without derivation.

13.2.3 Coefficients computation

We are interested in partial derivatives of the cost wrt the various weights, biases, and computed quantities. For this it's convenient to introduce a short-hand:

$$\delta_i^{[\ell]} = \frac{\partial C}{\partial z_i^{[\ell]}} \qquad \text{for } 1 \le i \le n_\ell \text{ and } 1 \le \ell < L. \tag{13.5}$$

Now applying the chain rule (for full derivation see the paper quoted above) we get, using $x \circ y$ for the pointwise (or *Hadamard*) vector-vector product $\{x_i y_i\}$:

- at the last level:

$$\delta^{[L-1]} = \sigma'\left(z^{[L-1]}\right) \circ \left(a^{[L]} - y\right)$$

- recursively for the earlier levels:

$$\delta^{[\ell]} = \sigma'\left(z^{[\ell]}\right) \circ \left(W^{[\ell+1]^t} \delta^{[\ell+1]}\right)$$

Input layer $\ell = 1$ starts with:		
$a^{(\ell)} = x$	network input	$n_{ell} \times b$
For layers $\ell = 1, \ldots, L$		
$a^{(\ell)}$	layer input	$n_\ell \times b$
$W^{(\ell)}$	weights	$n_{\ell+1} \times n_\ell$
$b^{(\ell)}$	biases	$n_{\ell+1} \times b$
$z^{(\ell)} \leftarrow W^{(\ell)} a^{(\ell)} + b^{(\ell)}$	biased product	$n_{\ell+1} \times b$
$a^{(\ell+1)} = y^{(\ell)} \leftarrow \sigma(z^{(\ell)})$	activated product	$n_{\ell+1} \times b$
The final output:		
$y = a^{(L+1)} = z^{(L)}$		$n_{L+1} \times b$
For layers $\ell = L, L-1, \ldots, 1$		
$D^{(\ell+1)} \leftarrow \mathrm{diag}(\sigma'(z^{(\ell)}))$		$n_{\ell+1} \times n_{\ell+1}$
$\delta^{(\ell)} \leftarrow \begin{cases} D^{L+1}(a^{L+1} - y) \\ D^{(\ell+1)} {W^{(\ell+1)}}^t \delta^{(\ell+1)} \end{cases}$ $\ell < L$		$n_{\ell+1} \times b$
$\Delta W^{(\ell)} \leftarrow \delta^{(\ell)} {a^{(\ell-1)}}^t$	weights update	$n_{\ell+1} \times n_\ell$
$w^{(\ell)} \leftarrow w^{(\ell)} - \Delta W^{(\ell)}$		
$\Delta b^{(\ell)} \equiv \delta^{(\ell)}$	bias update	$n_{\ell+1} \times b$

(13.6)

Figure 13.1: Deep Learning forward/backward passes.

- sensitivity wrt the biases:
$$\frac{\partial C}{\partial b_i^{[\ell]}} = \delta_i^{[\ell]}$$
- sensitivity wrt the weights:
$$\frac{\partial C}{\partial w_{ik}^{[\ell]}} = \delta_i^{[\ell]} a_k^{[\ell-1]}$$

Using the special form

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

gives

$$\sigma'(x) = \sigma(x)(1-\sigma(x)).$$

13.2.4 Algorithm

We now present the full algorithm in figure 13.1. Our network has layers $\ell = 1, \ldots, L$, where the parameter n_ℓ denotes the input size of layer ℓ.

Layer 1 has input x, and layer L has output y. Anticipating the use of minibatches, we let x, y denote a group of inputs/output of size b, so their sizes are $n_1 \times b$ and $n_{L+1} \times b$ respectively.

13.3 Computational aspects

In this section we will discuss high-performance computing aspects of DL. In a scalar sense, we argue for the the presence of the matrix-matrix product, which can be executed at high efficiency.

We will also discuss parallelism, focusing on

- Data parallelism, where the basic strategy is to split up the dataset;
- Model parallelism, where the basic strategy is to split up the model parameters; and
- Pipelining, where instructions can be executed in other orderings than in the naive model.

13.3.1 Weight matrix product

Both in applying the net, the forward pass, and in learning, the backward pass, we perform *matrix times vector product*s with the weights matrix. This operation does not have much cache reuse, and will therefore not have high performance; section 6.10.

On the other hand, if we bundle a number of datapoints – this is sometimes called a *mini-batch* – and operate on them jointly, our basic operation becomes the *matrix times matrix product*, which is capable of much higher performance; section 1.6.1.2.

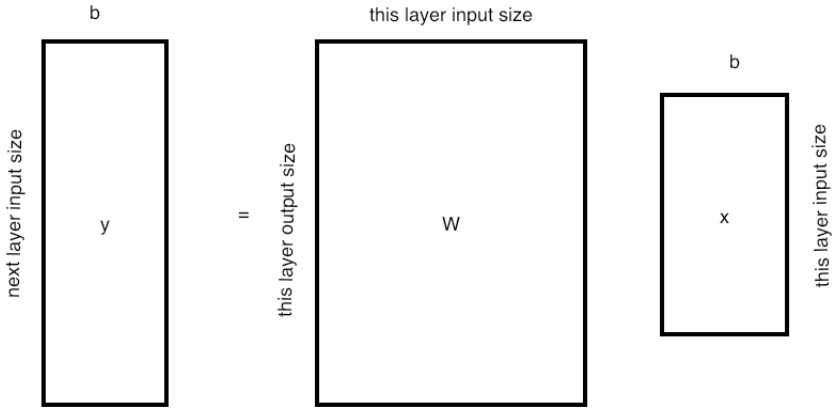


Figure 13.2: Shapes of arrays in a single layer.

We picture this in figure 13.2:

- the input batch and output batch consist of the same number of vectors;
- the weight matrix W has a number of rows equal to the output size, and a number of columns equal to the input size.

This importance of the `gemm` kernel (sections 1.6.1.2 and 7.4.1) has led people to develop dedicated hardware for it.

Another approach would be to use a special form for the weights matrix. In [138] approximation by a *Toeplitz matrix* is investigated. This has both advantages in space savings, and in that the product can be done through an *FFT*.

13.3.2 Parallelism in the weight matrix product

We can now consider the efficient computation of $N(x)$. We already remarked that matrix-matrix multiplication is an important kernel, but apart from that we can use parallel processing. Figure 13.3 gives two parallelization strategies.

In the first one, batches are divided over processes (or equivalently, multiple processes are working on independent batches simultaneously); We refer to this as *data parallelism*.

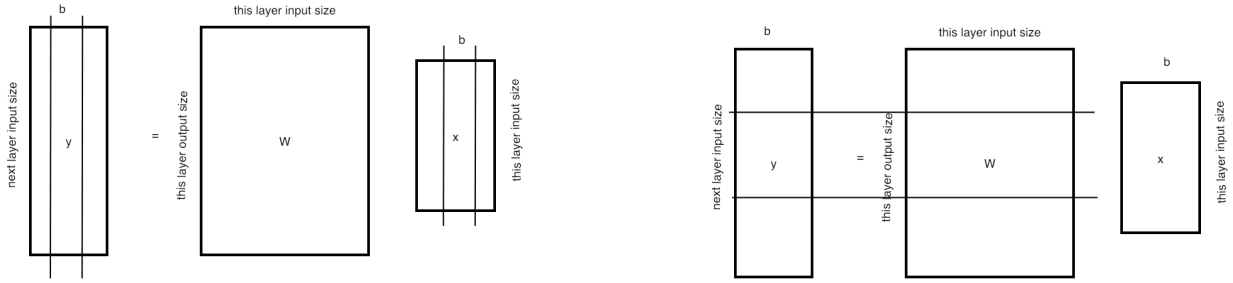


Figure 13.3: Partitioning strategies for parallel layer evaluation.

Exercise 13.1. Consider this scenario in a shared memory context. In this code:

for $b = 1, \ldots,$ batchsize
  for $i = 1, \ldots,$ outsize
    $y_{i,b} \leftarrow \sum_j W_{i,j} \cdot x_{j,b}$

assume that each thread computes part of the $1, \ldots,$ batchsize range.
Translate this to your favorite programming language. Do you store the input/output vectors by rows or columns? Why? What are the implications of either choice?

Exercise 13.2. Now consider the data parallelism in a distributed memory context, with each process working on a slice (block column) of the batches. Do you see an immediate problem?

There is a second strategy, referred to as *model parallelism*, where the model parameters, that is, the weights and biases, are distributed. As you can see in figure 13.3, this immediately implies that output vectors of the layer are computed distributely.

Exercise 13.3. Outline the algorithm of this second partitioning in a distributed memory context.

The choice between these strategies depends on whether the model is large, and the weight matrices need to be split up, or whether the number of inputs is large. Of course, a combination of these can also be made, where both the model and batches are distributed.

13.3.3 Weights update

The calculation of the weights update

$$\Delta W^{(\ell)} \leftarrow \delta^{(\ell)} {a^{(\ell-1)}}^t$$

is an *outer product* of rank b. It takes two vectors, and computes a low-rank matrix from them.

Exercise 13.4. Discuss the amount of (potential) data reuse in this operation, depending on the relation between n_ℓ and b. Assume $n_{\ell+1} \approx n_\ell$ for simplicity.

Exercise 13.5. Discuss the structure of the data movement involved, in both of the partitioning strategies of figure 13.3.

Apart from these aspects, this operation becomes even more interesting when we consider processing mini-batches in parallel. In that case every batch independently computes an update, and we need to average them. Under the assumption that each process compute a full ΔW, this becomes an *all-reduce*. This application of 'HPC techniques' was developed into the *Horovod* software [75, 174, 108]. In one example, considerable speedup was shown on a configuration involving 40 GPUs.

Another option would be delaying updates, or performing them asynchronously.

Exercise 13.6. Discuss implementing delayed or asynchronous updates in MPI.

13.3.4 Pipelining

A final type of parallelism can be achieved by applying pipelining over the layers. Sketch how this can improve the efficiency of the training stage.

13.3.5 Convolutions

Applying a convolution is equivalent to multiplying by a *Toeplitz matrix*. This has a lower complexity than a fully general matrix-matrix multiplication.

13.3.6 Sparse matrices

The weights matrix can be sparsified by ignoring small entries. This makes the *sparse matrix times dense matrix* product the dominant operation [70].

13.3.7 Hardware support

From the above, we conclude the importance of the `gemm` computational kernel. Dedicating a regular CPU exclusively to this purpose is a considerable waste of silicon and power. At the very least, using GPUs is an energy-efficient solution.

However, even more efficiency can be attained by using special purpose hardware. Here is an overview: `https://blog.inten.to/hardware-for-deep-learning-part-4-asic-96a542fe6a81` In a way, these special purpose processors are a re-incarnation of *systolic array*s.

13.3.8 Reduced precision

See section 3.8.4.2.

13.4 Stuff

Universal Approximation Theorem

Let $\varphi(\cdot)$ be a nonconstant,bounded, and monotonically-increasing continuous function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$. The space of continuous functions on I_m is denoted by $C(I_m)$. Then, given any function $f \in C(I_m)$ and $\varepsilon > 0$, there exists an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$, where $i = 1, \cdots, N$, such that we may define:

$$F(x) = \sum_{i=1}^{N} v_i \varphi\left(w_i^T x + b_i\right)$$

as an approximate realization of the function f where f is independent of φ; that is,

$$|F(x) - f(x)| < \varepsilon$$

for all $x \in I_m$. In other words, functions of the form $F(x)$ are dense in $C(I_m)$.

Can a NN approximate multiplication?
https://stats.stackexchange.com/questions/217703/can-deep-neural-network-approximat

Traditional neural network consists of linear maps and Lipschitiz activation function. As a composition of Lischitz continuous functions, neural network is also Lipschitz continuous, but multiplication is not Lipschitz continuous. This means that neural network cannot approximate multiplication when one of the x or y goes too large.

PART III

APPENDICES

This course requires no great mathematical sophistication. Mostly it assumes that you know the basics of linear algebra: what are matrices and vectors, and the most common operations on them.

In the following appendices we will cover some less common bits of theory that have been moved out of the main storyline of the preceeding chapters.

Chapter 14

Linear algebra

In this course it is assumed that you know what a matrix and a vector are, simple algorithms such as how to multiply them, and some properties such as invertibility of a matrix. This appendix introduces some concepts and theorems that are not typically part of a first course in linear algebra.

14.1 Norms

A *norm* is a way to generalize the concept of absolute value to multi-dimensional objects such as vectors and matrices. There are many ways of defining a norm, and there is theory of how different norms relate. Here we only give the basic concepts.

14.1.1 Vector norms

A norm is any function $n(\cdot)$ on a vector space V with the following properties:

- $n(x) \geq 0$ for all $x \in V$ and $n(x) = 0$ only for $x = 0$,
- $n(\lambda x) = |\lambda| n(x)$ for all $x \in V$ and $\lambda \in \mathbb{R}$.
- $n(x + y) \leq n(x) + n(y)$

For any $p \geq 1$, the following defines a vector norm:

$$|x|_p = \sqrt[p]{\sum_i |x_i|^p}.$$

Common norms are the 1-norm, 2-norm, and infinity norm:

- The 1-norm $\|\cdot\|_1$ is the sum of absolute values:

$$\|x\|_1 = \sum |x_i|.$$

- The 2-norm $\|\cdot\|_2$ is the root of the sum of squares:

$$\|x\|_2 = \sqrt{\sum x^2}.$$

- The infinity-norm $\|\cdot\|_\infty$ norm is defined as $\lim_{p\to\infty} \|\cdot\|_p$, and it is not hard to see that this equals

$$\|x\|_\infty = \max_i |x_i|.$$

14.1.2 Matrix norms

By considering a matrix of size n as a vector of length n^2, we can define the Frobenius matrix norm:

$$\|A\|_F = \sqrt{\sum_{i,j} |a_{ij}|^2}.$$

However, we will mostly look at *associated matrix norms*:

$$\|A\|_p = \sup_{\|x\|_p=1} \|Ax\|_p = \sup_x \frac{\|Ax\|_p}{\|x\|_p}.$$

From their definition, it then follows that

$$\|Ax\| \leq \|A\|\|x\|$$

for associated norms.

The following are easy to derive:

- $\|A\|_1 = \max_j \sum_i |a_{ij}|$, the maximum column-absolute sum;
- $\|A\|_\infty = \max_i \sum_j |a_{ij}|$, the maximum row-absolute sum.

By observing that $\|A\|_2 = \sup_{\|x\|_2=1} x^t A^t A x$, it is not hard to derive that $\|A\|_2$ is the maximal singular value of A, which is the root of the maximal eigenvalue of $A^t A$, which is the largest eigenvalue of A is A is symmetric.

The matrix *condition number* is defined as

$$\kappa(A) = \|A\| \, \|A^{-1}\|.$$

In the symmetric case, and using the 2-norm, this is the ratio between the largest and smallest *eigenvalue*; in general, it is the ratio between the largest and smallest *singular value*.

14.2 Gram-Schmidt orthogonalization

The Gram-Schmidt (GS) algorithm takes a series of vectors and inductively orthogonalizes them. This can be used to turn an arbitrary basis of a vector space into an orthogonal basis; it can also be viewed as transforming a matrix A into one with orthogonal columns. If Q has orthogonal columns, $Q^t Q$ is diagonal, which is often a convenient property to have.

The basic principle of the GS algorithm can be demonstrated with two vectors u, v. Suppose we want a vector v' so that u, v spans the same space as u, v', but $v' \perp u$. For this we let

$$v' \leftarrow v - \frac{u^t v}{u^t u} u.$$

It is easy to see that this satisfies the requirements.

Suppose we have an set of vectors $u_1, \ldots, u_n$ that we want to orthogonalize. We do this by successive applications of the above transformation:

For $i = 1, \ldots, n$:
  For $j = 1, \ldots i-1$:
    let $c_{ji} = u_j^t u_i / u_j^t u_j$
  For $i = 1, \ldots, n$:
    update $u_i \leftarrow u_i - u_j c_{ji}$

Often the vector v in the algorithm above is normalized; this adds a line

$u_i \leftarrow u_i / \|u_i\|$

to the algorithm. GS orthogonalization with this normalization, applied to the columns of a matrix, is also known as the *QR factorization*.

Exercise 14.1. Suppose that we apply the GS algorithm to the columns of a rectangular matrix A, giving a matrix Q. Prove that there is an upper triangular matrix R such that $A = QR$. (Hint: look at the c_{ji} coefficients above.) If we normalize the orthogonal vector in the algorithm above, Q has the additional property that $Q^t Q = I$. Prove this too.

The GS algorithm as given above computes the desired result, but only in exact arithmetic. A computer implementation can be quite inaccurate if the angle between v and one of the u_i is small. In that case, the Modified Gram-Schmidt (MGS) algorithm will perform better:

For $i = 1, \ldots, n$:
  For $j = 1, \ldots i-1$:
    let $c_{ji} = u_j^t u_i / u_j^t u_j$
    update $u_i \leftarrow u_i - u_j c_{ji}$

To contrast it with MGS, the original GS algorithm is also known as Classical Gram-Schmidt (CGS).

As an illustration of the difference between the two methods, consider the matrix

$$A = \begin{pmatrix} 1 & 1 & 1 \\ \epsilon & 0 & 0 \\ 0 & \epsilon & 0 \\ 0 & 0 & \epsilon \end{pmatrix}$$

where ϵ is of the order of the machine precision, so that $1+\epsilon^2 = 1$ in machine arithmetic. The CGS method proceeds as follows:

- The first column is of length 1 in machine arithmetic, so

$$q_1 = a_1 = \begin{pmatrix} 1 \\ \epsilon \\ 0 \\ 0 \end{pmatrix}.$$

- The second column gets orthogonalized as $v \leftarrow a_2 - 1 \cdot q_1$, giving

$$v = \begin{pmatrix} 0 \\ -\epsilon \\ \epsilon \\ 0 \end{pmatrix}, \quad \text{normalized:} \quad q_2 = \begin{pmatrix} 0 \\ -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \\ 0 \end{pmatrix}$$

- The third column gets orthogonalized as $v \leftarrow a_3 - c_1 q_1 - c_2 q_2$, where

$$\begin{cases} c_1 = q_1^t a_3 = 1 \\ c_2 = q_2^t a_3 = 0 \end{cases} \quad \Rightarrow v = \begin{pmatrix} 0 \\ -\epsilon \\ 0 \\ \epsilon \end{pmatrix}; \quad \text{normalized:} \quad q_3 = \begin{pmatrix} 0 \\ \frac{\sqrt{2}}{2} \\ 0 \\ \frac{\sqrt{2}}{2} \end{pmatrix}$$

It is easy to see that q_2 and q_3 are not orthogonal at all. By contrast, the MGS method differs in the last step:

- As before, $q_1^t a_3 = 1$, so

$$v \leftarrow a_3 - q_1 = \begin{pmatrix} 0 \\ -\epsilon \\ \epsilon \\ 0 \end{pmatrix}.$$

  Then, $q_2^t v = \frac{\sqrt{2}}{2}\epsilon$ (note that $q_2^t a_3 = 0$ before), so the second update gives

$$v \leftarrow v - \frac{\sqrt{2}}{2}\epsilon q_2 = \begin{pmatrix} 0 \\ \frac{\epsilon}{2} \\ -\frac{\epsilon}{2} \\ \epsilon \end{pmatrix}, \quad \text{normalized:} \quad \begin{pmatrix} 0 \\ \frac{\sqrt{6}}{6} \\ -\frac{\sqrt{6}}{6} \\ 2\frac{\sqrt{6}}{6} \end{pmatrix}$$

  Now all $q_i^t q_j$ are on the order of ϵ for $i \neq j$.

14.3 The power method

The vector sequence

$$x_i = Ax_{i-1},$$

where x_0 is some starting vector, is called the *power method* since it computes the product of subsequent matrix powers times a vector:

$$x_i = A^i x_0.$$

There are cases where the relation between the x_i vectors is simple. For instance, if x_0 is an eigenvector of A, we have for some scalar λ

$$Ax_0 = \lambda x_0 \qquad \text{and} \qquad x_i = \lambda^i x_0.$$

However, for an arbitrary vector x_0, the sequence $\{x_i\}_i$ is likely to consist of independent vectors. Up to a point.

Exercise 14.2. Let A and x be the $n \times n$ matrix and dimension n vector

$$A = \begin{pmatrix} 1 & 1 & & & \\ & 1 & 1 & & \\ & & \ddots & \ddots & \\ & & & 1 & 1 \\ & & & & 1 \end{pmatrix}, \qquad x = (0, \ldots, 0, 1)^t.$$

Show that the sequence $[x, Ax, \ldots, A^i x]$ is an independent set for $i < n$. Why is this no longer true for $i \geq n$?
Now consider the matrix B:

$$B = \left(\begin{array}{ccccc|ccccc} 1 & 1 & & & & & & & & \\ & \ddots & \ddots & & & & & & & \\ & & 1 & 1 & & & & & & \\ & & & 1 & & & & & & \\ \hline & & & & & 1 & 1 & & & \\ & & & & & & \ddots & \ddots & & \\ & & & & & & & 1 & 1 & \\ & & & & & & & & 1 & \end{array}\right), \qquad y = (0, \ldots, 0, 1)^t$$

Show that the set $[y, By, \ldots, B^i y]$ is an independent set for $i < n/2$, but not for any larger values of i.

While in general the vectors $x, Ax, A^2x, \ldots$ can be expected to be independent, in computer arithmetic this story is no longer so clear.

Suppose the matrix has eigenvalues $\lambda_0 > \lambda_1 \geq \cdots \lambda_{n-1}$ and corresponding eigenvectors u_i so that

$$Au_i = \lambda_i u_i.$$

Let the vector x be written as

$$x = c_0 u_0 + \cdots + c_{n-1} u_{n-1},$$

then

$$A^i x = c_0 \lambda_0^i u_i + \cdots + c_{n-1} \lambda_{n-1}^i u_{n-1}.$$

If we write this as

$$A^i x = \lambda_0^i \left[c_0 u_i + c_1 \left(\frac{\lambda_1}{\lambda_0} \right)^i + \cdots + c_{n-1} \left(\frac{\lambda_{n-1}}{\lambda_0} \right)^i \right],$$

we see that, numerically, $A^i x$ will get progressively closer to a multiple of u_0, the *dominant eigenvector*. Hence, any calculation that uses independence of the $A^i x$ vectors is likely to be inaccurate. Section 5.5.9 discusses iterative methods that can be considered as building an orthogonal basis for the span of the power method vectors.

14.4 Nonnegative matrices; Perron vectors

If A is a nonnegative matrix, the maximal eigenvalue has the property that its eigenvector is nonnegative: this is the the *Perron-Frobenius theorem.*

Theorem 5 *If a nonnegative matrix A is irreducible, its eigenvalues satisfy*

- *The eigenvalue α_1 that is largest in magnitude is real and simple:*

$$\alpha_1 > |\alpha_2| \geq \cdots .$$

- *The corresponding eigenvector is positive.*

The best known application of this theorem is the Googe *PageRank* algorithm; section 10.4.

14.5 The Gershgorin theorem

Finding the eigenvalues of a matrix is usually complicated. However, there are some tools to estimate eigenvalues. In this section you will see a theorem that, in some circumstances, can give useful information on eigenvalues.

Let A be a square matrix, and x, λ an eigenpair: $Ax = \lambda x$. Looking at one component, we have

$$a_{ii}x_i + \sum_{j\neq i} a_{ij}x_j = \lambda x_i.$$

Taking norms:

$$(a_{ii} - \lambda) \leq \sum_{j\neq i} |a_{ij}| \left|\frac{x_j}{x_i}\right|$$

Taking the value of i for which $|x_i|$ is maximal, we find

$$(a_{ii} - \lambda) \leq \sum_{j\neq i} |a_{ij}|.$$

This statement can be interpreted as follows:

The eigenvalue λ is located in the circle around a_{ii} with radius $\sum_{j\neq i} |a_{ij}|$.

Since we do not know for which value of i $|x_i|$ is maximal, we can only say that there is *some* value of i such that λ lies in such a circle. This is the *Gershgorin theorem.*

Theorem 6 *Let A be a square matrix, and let D_i be the circle with center a_{ii} and radius $\sum_{j\neq i} |a_{ij}|$, then the eigenvalues are contained in the union of circles $\cup_i D_i$.*

We can conclude that the eigenvalues are in the interior of these discs, if the constant vector is not an eigenvector.

14.6 Householder reflectors

In some contexts the question comes up how to transform one subspace into another. *Householder reflectors* are in a sense the minimal solution to this. Consider a unit vector u, and let

$$H = I - 2uu^t.$$

For this matrix we have $Hu = -u$, and if $u \perp v$, then $Hv = v$. In other words, the subspace of multiples of u is flipped, and the orthogonal subspace stays invariant.

Now for the original problem of mapping one space into another. Let the original space be spanned by a vector x and the resulting by y, then note that

$$\begin{cases} x = (x+y)/2 + (x-y)/2 \\ y = (x+y)/2 - (x-y)/2. \end{cases}$$

In other words, we can map x into y with the reflector based on $u = (x-y)/2$.

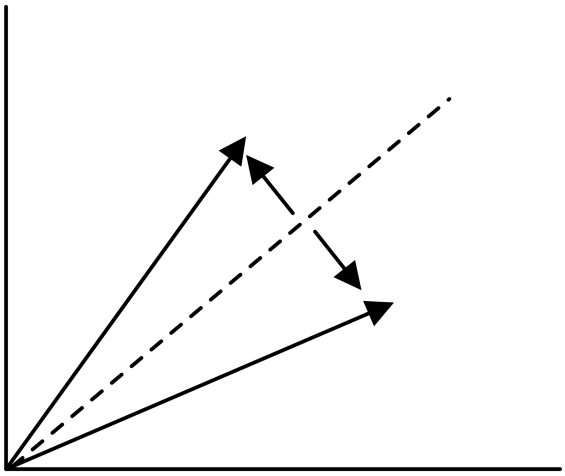


Figure 14.1: Householder reflector.

We can generalize Householder reflectors to a form

$$H = I - 2uv^t.$$

The matrices L_i used in LU factorization (see section 5.3) can then be seen to be of the form $L_i = I - \ell_i e_i^t$ where e_i has a single one in the i-th location, and ℓ_i only has nonzero below that location. That form also makes it easy to see that $L_i^{-1} = I + \ell_i e_i^t$:

$$(I - uv^t)(I + uv^t) = I - uv^t uv^t = 0$$

if $v^t u = 0$.

Chapter 15

Complexity

At various places in this book we are interested in how many operations an algorithm takes. It depends on the context what these operations are, but often we count additions (or subtractions) and multiplications. This is called the *arithmetic* or *computational complexity* of an algorithm. For instance, summing n numbers takes $n-1$ additions. Another quantity that we may want to describe is the amount of space (computer memory) that is needed. Sometimes the space to fit the input and output of an algorithm is all that is needed, but some algorithms need temporary space. The total required space is called the *space complexity* of an algorithm.

Both arithmetic and space complexity depend on some description of the input, for instance, for summing an array of numbers, the length n of the array is all that is needed. We express this dependency by saying 'summing an array of numbers has time complexity $n-1$ additions, where n is the length of the array'.

The time (or space) the summing algorithm takes is not dependent on other factors such as the values of the numbers. By contrast, some algorithms such as computing the greatest common divisor of an array of integers *can* be dependent on the actual values. In the section on sorting 9 you will see both kinds.

Exercise 15.1. What is the time and space complexity of multiplying two square matrices of size $n \times n$? Assume that an addition and a multiplication take the same amount of time.

Often we aim to simplify the formulas that describe time or space complexity. For instance, if the complexity of an algorithm is $n^2 + 2n$, we see that for $n > 2$ the complexity is less than $2n^2$, and for $n > 4$ it is less than $(3/2)n^2$. On the other hand, for all values of n the complexity is at least n^2. Clearly, the quadratic term n^2 is the most important, and the linear term n becomes less and less important by ratio. We express this informally by saying that the complexity is quadratic in n as $n \to \infty$: there are constants c, C so that for n large enough the complexity is at least cn^2 and at most Cn^2.

This is expressed for short by saying that the complexity is of *order* n^2, written as $O(n^2)$ as $n \to \infty$. In chapter 4 you will see phenomena that can be described as orders of a parameter that goes to zero. In that case we write for instance $f(h) = O(h^2)$ as $h \downarrow 0$, meaning that f is bounded by ch^2 and Ch^2 for certain constants c, C and h small enough.

15.1 Formal definitions

Formally, we distinguish the following complexity types:

- Big-Oh complexity. This states the existence of an upperbound on the absolute magnitude:

$$g(x) = O(f(x)) \equiv \exists_{C>0,x_0>0} \forall_{x>x_0} : |g(x)| < Cf(x)$$

- Little-oh complexity. This states a function is essentially smaller than another:

$$g(x) = o(f(x)) \equiv \forall_{C>0} \exists_{x_0>0} \forall_{x>x_0} : |g(x)| < Cf(x)$$

- Big-Theta complexity. This is like Big-Oh, except that now we have upper and lower bounds:

$$g(x) = \Theta(f(x)) \equiv \exists_{C>c>0,x_0>0} \forall_{x>x_0} : cf(x) < g(x) < Cf(x)$$

- Finally: Big-Omega complexity states that a function is at least as large as another:

$$g(x) = \Omega(f(x)) \equiv \exists_{C>0,x_0>0} \forall_{x>x_0} : g(x) > Cf(x)$$

15.2 The Master Theorem

For many operations, complexity is easily expressed recursively. For instance, the *complexity of quicksort* obeys a recurrence

$$T(n) = 2T(n/2) + O(n),$$

expressing that sorting an array of n numbers involves twice sorting $n/2$ numbers, plus the splitting step, which is linear in the array size.

The so-called *master theorem* of complexity allows you to convert these recursive expression to a closed form formula.

Suppose the function T satisfies

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

with $a, b \geq 1$ and $f(\cdot)$ asymptotically positive. Then there are three cases:

1. If

$$f(n) = O\left(n^{\log_b(a)-\epsilon}\right) \quad \text{then} \quad T(n) = \Theta\left(n^{\log_b(a)}\right)$$

2. If

$$f(n) = O\left(n^{\log_b(a)}\right) \quad \text{then} \quad T(n) = \Theta\left(n^{\log_b(a)} \log(n)\right)$$

3. If

$$f(n) = O\left(n^{\log_b(a)+\epsilon}\right) \quad \text{then} \quad T(n) = \Theta\left(f(n)\right)$$

Some examples follow.

15.2.1 Quicksort and mergesort

The *quicksort* and *mergesort* algorithms recursively divide work into two (hopefully) equal halves, preceeded/followed by a linear splitting/merging step. Thus their complexity satisfies

$$T(n) = 2T(n/2) + O(n)$$

giving, with $a = b = 2$ and $f(n) = n$, by case 2:

$$T(n) = \Theta(n \log n).$$

15.2.2 Matrix-matrix multiplication

We already know that the ordinary triple-loop formulation of matrix-matrix multiplication has a complexity of $O(n^3)$. Multiplying matrices by recursive bisection into 2×2 blocks gives

$$T(n) = 8T(n/2) + O(n^2).$$

Substituting $a = 8, b = 2$ gives $\log_b(a) = 3$ so case 1 applies with $\epsilon = 1$, and we find

$$T(n) = \Theta(n^3).$$

Slightly more interesting, the *Strassen* algorithm [182] has $a = 7$, giving an essentially lower complexity of $O(n^{\log 7}) \approx O(n^{2.81})$. There are other algorithms like this [160], with slightly lower complexities, though all still well above $O(n^2)$.

15.3 Little-oh complexity

In most algorithms we are interested in big-oh complexity. Computing little-oh complexity is more rare.

As an example, considering factoring large integers, an important part of *cryptography*. Naively, to find factors of a number N, we would at worst have to try all numbers up to $\sqrt{N}$. One of the best algorithms is the *quadratic sieve*, which has a complexity of $O(e^{\sqrt{\ln n \ln\ln n}})$.

Is this better than $O(\sqrt{N})$? Yes, because

$$e^{\sqrt{\ln n \ln\ln n}} = o(\sqrt{n}).$$

Exercise 15.2. Prove this. Hint: write $\sqrt{n} = e^{\ln \sqrt{n}}$.

15.4 Amortized complexity

.

There are cases where you are interested in doing a sequence of operations, where the cost of each can vary widely. In that case, looking at the worst operation is valid, but from a practical point of view not the most relevant. For this, the concept of *amortized complexity* has been coined [183].

A common example of amortized complexity is dynamically extending an array. Assume we have allocated space for n contiguous elements, and we want to add a new element to the array, keeping the data contiguous. The following is the space-minimal solution:

1. Allocate $n + 1$ elements;
2. copy the existing data over;
3. write the new element;
4. de-allocate the old array.

Exercise 15.3. What is the complexity of doing this operation repeatedly? Ignore all cost of creating the arrays, and consider only the copy operation, assuming a unit time per element.

A more efficient scheme is used in the run-time libraries of such languages as C++:

1. Given allocated space for n elements, and wanting to store an $n + 1$st element:
2. Allocate $2n$ elements;
3. Copy data over, write the new element;
4. De-allocate the old array, and:
5. For the next $n - 1$ elements, write into the unused portions of the new array.

This solution is not minimal in space: if you never fill up the $2n$ elements, some memory will be wasted. However, by the above measure of the cost of writing elements it is optimal.

Exercise 15.4. Show that the cost of storing n elements by this scheme is $O(n)$. What is the actual multiplicative constant?

Since writing n elements has complexity $O(n)$, we say that the *amortized complexity* of this scheme is $O(1)$, which is optimal up to a constant.

Chapter 16

Partial Differential Equations

Partial Differential Equations are the source of a large fraction of HPC problems. Here is a quick derivation of two of the most important ones.

16.1 Partial derivatives

Derivatives of a function $u(x)$ are a measure of the rate of change. Partial derivatives to the same, but for a function $u(x,y)$ of two variables. Notated u_x and u_y, these *partial derivates* indicate the rate of change if only one variable changes and the other stays constant.

Formally, we define u_x, u_y by:

$$u_x(x,y) = \lim_{h\to 0} \frac{u(x+h,y)-u(x,y)}{h}, \quad u_y(x,y) = \lim_{h\to 0} \frac{u(x,y+h)-u(x,y)}{h}$$

16.2 Poisson or Laplace Equation

Let T be the temperature of a material, then its heat energy is proportional to it. A segment of length Δx has heat energy $Q = c\Delta x \cdot u$. If the heat energy in that segment is constant

$$\frac{\delta Q}{\delta t} = c\Delta x \frac{\delta u}{\delta t} = 0$$

but it is also the difference between inflow and outflow of the segment. Since flow is proportional to temperature differences, that is, to u_x, we see that also

$$0 = \left.\frac{\delta u}{\delta x}\right|_{x+\Delta x} - \left.\frac{\delta u}{\delta x}\right|_{x}$$

In the limit of $\Delta x \downarrow 0$ this gives $u_{xx} = 0$, which is called the *Laplace equation*. If we have a source term, for instance corresponding to externally applied heat, the equation becomes $u_{xx} = f$, which is called the *Poisson equation*.

16.3 Heat Equation

Let T be the temperature of a material, then its heat energy is proportional to it. A segment of length Δx has heat energy $Q = c\Delta x \cdot u$. The rate of change in heat energy in that segment is

$$\frac{\delta Q}{\delta t} = c\Delta x \frac{\delta u}{\delta t}$$

but it is also the difference between inflow and outflow of the segment. Since flow is proportional to temperature differences, that is, to u_x, we see that also

$$\frac{\delta Q}{\delta t} = \frac{\delta u}{\delta x}\Big|_{x+\Delta x} - \frac{\delta u}{\delta x}\Big|_{x}$$

In the limit of $\Delta x \downarrow 0$ this gives $u_t = \alpha u_{xx}$.

16.4 Steady state

The solution of an IBVP is a function $u(x,t)$. In cases where the forcing function and the boundary conditions do not depend on time, the solution will converge in time, to a function called the *steady state* solution:

$$\lim_{t\to\infty} u(x,t) = u_{\text{steadystate}}(x).$$

This solution satisfies a BVP, which can be found by setting $u_t \equiv 0$. For instance, for the heat equation

$$u_t = u_{xx} + q(x)$$

the steady state solution satisfies $-u_{xx} = q(x)$.

Chapter 17

Taylor series

Taylor series expansion is a powerful mathematical tool. In this course it is used several times in proving properties of numerical methods.

The Taylor expansion theorem, in a sense, asks how well functions can be approximated by polynomials, that is, for a given function f, can we find coefficients c_i with $i = 1, \ldots, n$ so that

$$f(x) \approx c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n.$$

This question obviously needs to be refined. What do we mean by 'approximately equal'? This approximation formula can not hold for all functions f and all x: the function $\sin x$ is bounded for all x, but any polynomial is unbounded for $x \to \pm\infty$, so any polynomial approximation to the $\sin x$ function is unbounded. Clearly we can only approximate on an interval.

We will show that a function f with sufficiently many derivatives can be approximated as follows: if the n-th derivative $f^{(n)}$ is continuous on an interval I, then there are coefficients $c_0, \ldots, c_{n-1}$ such that

$$\forall_{x \in I} : |f(x) - \sum_{i<n} c_i x^i| \leq c M_n \qquad \text{where } M_n = \max_{x \in I} |f^{(n)}(x)|$$

It is easy to get inspiration for what these coefficients should be. Suppose

$$f(x) = c_0 + c_1 x + c_2 x^2 + \cdots$$

(where we will not worry about matters of convergence and how long the dots go on) then filling in

$$x = 0 \text{ gives } c_0 = f(0).$$

Next, taking the first derivative

$$f'(x) = c_1 + 2c_2 x + 3c_3 x^2 + \cdots$$

and filling in

$$x = 0 \text{ gives } c_1 = f'(0).$$

From the second derivative

$$f''(x) = 2c_2 + 6c_3x + \cdots$$

so filling in $x = 0$ gives

$$c_2 = f''(0)/2.$$

Similarly, in the third derivative

filling in $x = 0$ gives $c_3 = \frac{1}{3!}f^{(3)}(0)$.

Now we need to be a bit more precise. Cauchy's form of Taylor's theorem says that

$$f(x) = f(a) + \frac{1}{1!}f'(a)(x-a) + \cdots + \frac{1}{n!}f^{(n)}(a)(x-a)^n + R_n(x)$$

where the 'rest term' R_n is

$$R_n(x) = \frac{1}{(n+1)!}f^{(n+1)}(\xi)(x-a)^{n+1} \quad \text{where } \xi \in (a,x) \text{ or } \xi \in (x,a) \text{ depending.}$$

If $f^{(n+1)}$ is bounded, and $x = a + h$, then the form in which we often use Taylor's theorem is

$$f(x) = \sum_{k=0}^{n} \frac{1}{k!}f^{(k)}(a)h^k + O(h^{n+1}).$$

We have now approximated the function f on a certain interval by a polynomial, with an error that decreases geometrically with the inverse of the degree of the polynomial.

For a proof of Taylor's theorem we use integration by parts. First we write

$$\int_a^x f'(t)dt = f(x) - f(a)$$

as

$$f(x) = f(a) + \int_a^x f'(t)dt$$

Integration by parts then gives

$$\begin{aligned} f(x) &= f(a) + [xf'(x) - af'(a)] - \textstyle\int_a^x tf''(t)dt \\ &= f(a) + [xf'(x) - xf'(a) + xf'(a) - af'(a)] - \textstyle\int_a^x tf''(t)dt \\ &= f(a) + x\textstyle\int_a^x f''(t)dt + (x-a)f'(a) - \int_a^x tf''(t)dt \\ &= f(a) + (x-a)f'(a) + \textstyle\int_a^x (x-t)f''(t)dt \end{aligned}$$

Another application of integration by parts gives

$$f(x) = f(a) + (x-a)f'(a) + \frac{1}{2}(x-a)^2 f''(a) + \frac{1}{2}\int_a^x (x-t)^2 f'''(t)dt$$

Inductively, this gives us Taylor's theorem with

$$R_{n+1}(x) = \frac{1}{n!}\int_a^x (x-t)^n f^{(n+1)}(t)dt$$

By the mean value theorem this is

$$\begin{aligned} R_{n+1}(x) &= \frac{1}{(n+1)!} f^{(n+1)}(\xi) \int_a^x (x-t)^n f^{(n+1)}(t)dt \\ &= \frac{1}{(n+1)!} f^{(n+1)}(\xi)(x-a)^{n+1} \end{aligned}$$

Chapter 18

Minimization

18.1 Descent methods

We consider a multivariate function $f : R^N \rightarrow$ and the problem of finding the vector $\bar{x}$ for which is attains its minimum. Even for smooth functions we are immediately faced with the fact that there are local and global minima; for now we satisfy ourselves with finding a local minimum.

Rather than finding the minimum in one large calculation, we use an iterative strategy where we start with a point $\bar{x}$, updating it to $\bar{x} + \bar{h}$, and repeating this process. The update vector $\bar{h}$ is found through a *line search* strategy: we settle on a *search direction* $\bar{h}$, find a *step size* τ along that direction, and update

$$\bar{x} \leftarrow \bar{x} + \tau\bar{h}.$$

18.1.1 Steepest descent

Using the multi-dimensional form of the Taylor expansion formula (section 17) we can write any sufficiently smooth function as

$$f(\bar{x} + \bar{h}) = f(\bar{x}) + \bar{h}^t \nabla f + \cdots.$$

it is not hard to see that choosing $\bar{h} = -\nabla f$ gives the most minimization, so we set

$$x_{\text{new}} \equiv x - \tau\nabla f.$$

This method is called *gradient descent* or *steepest descent*.

For the new function value this gives

$$f(\bar{x} - \tau\nabla f) \approx f(\bar{x}) - \tau\|\nabla f\|^2$$

so for τ small enough this makes the new function value both positive, and less than $f(\bar{x})$.

The *step size* τ can be computed for quadratic functions f, and approximated otherwise:

$$f(\bar{x} + \tau\bar{h}) = f(\bar{x}) + \tau\bar{h}^t \nabla f + \frac{\tau^2}{2} h^t(\nabla \cdot \nabla f)h + \cdots.$$

Ignoring higher order terms and setting $\delta f/\delta\tau = 0$ gives

$$\tau = -\bar{h}^t \nabla f / h^t (\nabla \cdot \nabla f) h.$$

Another strategy for finding a suitable step size is known as *backtracking*:

- Start with a default step size, for instance $\tau \equiv 1$.
- Then until $f(\bar{x} + \tau\bar{h}) < f(\bar{x})$ do

$$\tau \leftarrow \tau/2.$$

18.1.2 Stochastic gradient descent

While the gradient is the optimal search direction in any given step, overall it need not be the best. For instance, for quadratic problems, which includes many problems from PDEs, a better choice is to orthogonalize search directions. On the other hand, in ML applications, *Stochastic Gradient Descent (SGD)* is a good choice, where the coordinate directions are used as search directions. In this case

$$f(x + \tau e_i) = f(x) + \tau \frac{df}{de_i} + \frac{\tau^2}{2} \frac{\delta^2 f}{\delta e_i^2}$$

Then:

$$\tau = -(\nabla f)_i / \frac{\delta^2 f}{\delta e_i^2}.$$

18.1.3 Code

18.1.3.1 Preliminaries

We declare a `vector` class that is a standard vector, with operations defined on it such as addition, but also rotation.

There is a derived class `unit_vector` that does the obvious.

18.1.3.2 Framework

We start by defining functions as a pure virtual class, meaning that any function needs to support the methods mentioned here:

MISSING SNIPPET functiondef in codesnippetsdir=snippets.code

Using such a function it becomes possible to define various update steps. For instance, the steepest descent step uses the gradient:

MISSING SNIPPET steepestdescentstep in codesnippetsdir=snippets.code

On the other hand, stochastic descent is based on unit vectors:

MISSING SNIPPET stochasticdescentstep in codesnippetsdir=snippets.code

18.1.3.3 *Sanity tests*

Using steepest descent on a circle gives convergence in one step:

```
Code:

MISSING SNIPPET descentcircle in
codesnippetsdir=snippets.code
```

```
Output:
[code/minimization] descentcircle:

Starting point (1,1); with value: 2
  becomes (0,0); with value: 0
```

On the other hand, steepest descent on an ellipse takes a number of iterations:

```
Code:

MISSING SNIPPET descentellipse in
codesnippetsdir=snippets.code
```

```
Output:
[code/minimization] descentellipse:

Starting point (1.0000,1.0000);
    ↪with value: 101.0000
  becomes (0.8991,-0.0090); with
    ↪value: 0.8165
Starting point (0.8991,-0.0090);
    ↪with value: 0.8165
  becomes (0.0736,0.0736); with
    ↪value: 0.5466
Starting point (0.0736,0.0736);
    ↪with value: 0.5466
  becomes (0.0661,-0.0007); with
    ↪value: 0.0044
Starting point (0.0661,-0.0007);
    ↪with value: 0.0044
  becomes (0.0054,0.0054); with
    ↪value: 0.0030
Starting point (0.0054,0.0054);
    ↪with value: 0.0030
  becomes (0.0049,-0.0000); with
    ↪value: 0.0000
```

18.2 Newton's method

Newton's method (or the *Newton-Raphson method*) is the iterative procedure

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$

for finding a *zero of a function* f, that is, a value x for which $f(x) = 0$. It requires knowledge of the derivative f' of the function, and it can be justified from figures such as 18.1.

Iteratively:

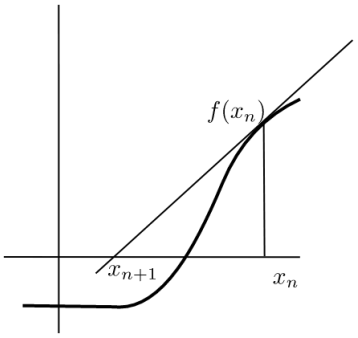


Figure 18.1: One step in the one-dimensional Newton method.

Another justification comes from minimization: if a function f is twice differentiable, we can write

$$f(x+h) = f(x) + h^t \nabla f + \frac{1}{2} h^t (\nabla^2 f) h$$

and the minimum is attained at

$$x_{\text{new}} = x - (\nabla^2 f)^{-1} \nabla f.$$

This is equivalent to finding a zero of the gradient:

$$0 = \nabla f(x+h) = \nabla f(x) + h^t (\nabla^2 f(x)) h.$$

Exercise 18.1. Let $f(x_1, x_2) = (10x_1^2 + x_2^2)/2 + 5\log(1 + e^{-x_1 - x_2})$. How fast do gradient descent and the Newton's method converge? To get insight in their differing behaviors, plot a number of iterates against level curves of the function.

This exercise gives some idea of that is wrong with gradient descent: it always steps perpendicular to the current level curve. However, the minimum does not necessarily lie in that direction.

Without proof we state:

- The Newton method will converge to the zero if the starting point of the iteration is close enough to the zero, and if the function is differentiable in the zero.
- In many circumstances it shows very fast convergence: convergence is quadratic, which is also described as the number of correct digits doubling in each iteration.
- For many functions Newton's method will not converge, but it is possible to obtain convergence by introducing damping, or doing an inexact update:

  $$x_{n+1} = x_n - \alpha f(x_n)/f'(x_n)$$

  where $\alpha < 1$.

Exercise 18.2. It is possible for Newton's method to be in a cycle. Suppose this is a cycle of length two:

$$x_0 \to x_1 \to x_2 = x_0.$$

If you write out the equations for this cycle you'll find a differential equation for f. What is the solution? Why doesn't the Newton method converge for this function?

In multiple dimensions, that is, with a function $f : \mathbb{R}^N \to \mathbb{R}$ Newton's method becomes an iterated *linear system solution* :

$$\bar{x}_{n+1} = \bar{x}_n - F(\bar{x}_n)^{-1} f(\bar{x}_n)$$

where F is the *Jacobian* of f.

Since Newton's method is an iterative process we do not need the solution of this linear system to full accuracy, so inexact solutions, for instance through *iterative solution*, is often feasible.

18.2.1 Inexact Newton's method

There is a variety of reasons why Newton's method would not converge. (In fact, plotting the regions from where it does converge will, for suitable functions, give nice *fractals*.) For this reason, in practice an *inexact Newton's method* is used. The inexactness comes in two ways:

- Instead of the 'optimal' step length we use a fraction. This is often chosen through 'backtracking', and a choice is adopted for which at least some decrease in function value is observed.
- Instead of the inverse of the Jacobian we use an approximation of that operator. For instance, we could compute the Jacobian (often an expensive process) and then use it for multiple Newton steps.

A combination of these techniques can be shown to give guaranteed converence [203].

Chapter 19

Random numbers

Random number generation is useful, for generating random test data, or in *Monte Carlo simulation*; see chapter 12.

Here we discuss Random Number Generators (RNGs) in general, their use in programming languages, and the problems of parallel random number generationn.

19.1 Random Number Generation

Random numbers are often used in simulations as some examples below will show. True random numbers are very hard to obtain: they could be generated by measuring quantum processes such as radioactive particles. Starting with the *Intel Ivy Bridge*, Intel's processors have a hardware random number generator based on thermal noise [95].)

The most common solution is to use *pseudo-random numbers*. This means that we use a deterministic mathematical process, that is sufficiently irregular that for practical purposes no order can be found in it.

19.1.1 Sequential random number generators

An easy way to generate random numbers (we leave off the 'pseudo' qualification) is to use *linear congruential* generators (for all you ever need to know about random numbers, see Knuth [122]), recurrences of the form

$$x_{k+1} = (ax_k + b) \mod m.$$

This sequence is periodic, since it consists of nonnegative integers at most $m - 1$, and with period m under certain conditions. A typical period is 2^{31}. The starting point x_0 of the series is known as the 'seed'. Software for random numbers often lets you specify the seed. To get reproducible results you would run your program with the same seed multiple times; to get random behavior over multiple runs of your program you could for instance derive the seed from clock and calendar functions.

Linear congruential generators may have some amount of correlation between lower bits. A different principle of generating random numbers is the *lagged Fibonacci* random number generator

$$X_i = X_{i-p} \otimes X_{i-q}$$

where p, q are the lag parameter, and $\otimes$ is any binary operation, such as addition or multiplication modulo M.

The main problems with lagged Fibonacci generators are:

- They require setting $\max(p, q)$ initial values, and their randomness is sensitive to these choices;
- They theory is not as developed as for congruential generators, so their is a greater reliance on statistical tests to evaluate their 'randomness'.

19.2 Random numbers in programming languages

19.2.1 C

There is an easy (but not terribly great) RNG that works the same in C and C++.

```
#include <random>
using std::rand;
float random_fraction =
    (float)rand()/(float)RAND_MAX;
```

The function `rand` yields an `int` – a different one every time you call it – in the range from zero to `RAND_MAX`. Using scaling and casting you can then produce a fraction between zero and one with the above code.

If you run your program twice, you will twice get the same sequence of random numbers. That is great for debugging your program but not if you were hoping to do some statistical analysis. Therefore you can set the *random number seed* from which the random sequence starts by the `srand` function. Example:

```
srand(time(NULL));
```

seeds the random number generator from the current time. This call should happen only once, typically somewhere high up in your main.

19.2.1.1 Problems with the C random generator

- The random number generator has a period of 2^{15}, which may be small.
- There is only one generator algorithm, which is implementation-dependent, and has no guarantees on its quality.
- There are no mechanisms fort ransforming the sequence to a range. The common idiom

  ```
  int under100 = rand() % 100
  ```

  is biased to small numbers. Figure 19.1 shows this for a generator with period 7 taken modulo 3.

19.2.1.2 Other generators

There are other RNGs for C, coming from Unix system calls.

- `drand48`: obsolete in *System V 3*, replaced by `rand`.
- `rand_r`: a *thread-safe* variant of `rand`. Marked obsolete in *POSIX 1-2008*
- `random`: a more modern version of `rand`.
- `random_r`: a thread-safe variant of `random`.

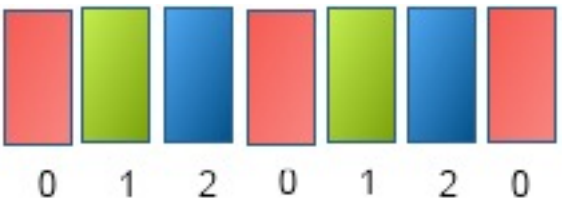


Figure 19.1: Low number bias of a random number generator taken module.

19.2.2 C++

The Standard Template Library (STL) has a *random number generator* that is more general and more flexible than the C version.

- There are several generators that give uniformly distributed numbers;
- then there are distributions that translate this to non-uniform or discrete distributions.

First you declare an engine; later this will be transformed into a distribution:

```
std::default_random_engine generator;
```

This generator will start at the same value every time. You can seed it:

```
std::random_device r;
std::default_random_engine generator{ r() };
```

Next, you need to declare the distribution. For instance, a uniform distribution between given bounds:

```
std::uniform_real_distribution<float> distribution(0.,1.);
```

A roll of the dice would result from:

```
std::uniform_int_distribution<int> distribution(1,6);
```

19.2.2.1 Random floats

```
// seed the generator
std::random_device r;
// std::seed_seq ssq{r()};
// and then passing it to the engine does the same

// set the default random number generator
std::default_random_engine generator{r()};

// distribution: real between 0 and 1
std::uniform_real_distribution<float> distribution(0.,1.);

cout << "first rand: " << distribution(generator) << endl;
```

19.2.2.2 Dice throw

```
// set the default generator
std::default_random_engine generator;

// distribution: ints 1..6
std::uniform_int_distribution<int> distribution(1,6);

// apply distribution to generator:
int dice_roll = distribution(generator);
  // generates number in the range 1..6
```

19.2.2.3 Poisson distribution

Another distribution is the *Poisson distribution*:

```
std::default_random_engine generator;
float mean = 3.5;
std::poisson_distribution<int> distribution(mean);
int number = distribution(generator);
```

19.2.3 Fortran

In this section we briefly discuss the Fortran *random number generator*. The basic mechanism is through the library subroutine `random_number`, which has a single argument of type `REAL` with `INTENT(OUT)`:

```
real(4) :: randomfraction
call random_number(randomfraction)
```

The result is a random number from the uniform distribution on $[0, 1)$.

Setting the *random seed* is slightly convoluted. The amount of storage needed to store the seed can be processor and implementation-dependent, so the routine `random_seed` can have three types of named argument, exactly one of which can be specified at any one time. The keyword can be:

- `SIZE` for querying the size of the seed;
- *PUT* for setting the seed; and
- *GET* for querying the seed.

A typical fragment for setting the seed would be:

```
integer :: seedsize
integer,dimension(:),allocatable :: seed

call random_seed(size=seedsize)
allocate(seed(seedsize))
seed(:) = ! your integer seed here
call random_seed(put=seed)
```

19.2.4 Python

Python has a random module:

```
import random
x = random.random()
i = random.randint(lo,hi)
```

19.3 Parallel random number generation

Random number generation is problematic in parallel. To see this, consider a parallel process that uses a random number generator on each subprocess, and consider a single processor emulating the parallel process. Now this single process in effect has a random number generator that consists of interleaving the parallel generator results. This means that, if we use the same generator in all parallel processes, the effective generator over the whole process will produce stretches of identical values.

There are various ways out.

19.3.1 Manager-worker generator

We can generate the random numbers centrally. In shared memory that could mean making its operation atomic. This may introduce a serious bottleneck.

Exercise 19.1. Critical sections are usually justified if the work spent there is of lower order than the parallel work. Why does that argument not apply here.

Another solution would be to have one thread or process that generates the random numbers and distributes them to the other processes. Doing this on a number-by-number basis causes considerable overhead. Instead, it would be possible for the generator process to distribute blocks of numbers. However, the manner in which this is done may again cause correlation between processes.

19.3.2 Sequence splitting solutions

A better solution is to set up independent generators with parameter choices that guarantee statistical randomness. This is not simple. For instance, if two sequences $x_i^{(1)}, x_i^{(2)}$ have the same values of a, b, m, and their starting points are close together, the sequences will be strongly correlated. Less trivial examples of correlation exist.

Various techniques for random number generation exist, such as using two sequences, where one generates the starting points for the other sequence, which is the one actually used in the simulation. Software for parallel random number generator can be found at http://sprng.cs.fsu.edu/ [146].

If it is possible given x_i to compute x_{i+k} cheaply, one use a leapfrogging technique, where k processes have disjoint series $i \mapsto x_{s_k+ik}$ where x_{s_k} is the starting point for the k-th series.

19.3.3 Blocked random number generators

Some random number generators (see [135]) allow you to calculate a value that is many iteration away from the seed. You could then take the block of values from the seed to that iteration and give it to one processor. Similarly, each processor would get a contiguous block of iterations of the generator.

19.3.4 Keyed random number generators

Some random number generators are not recursive, but allow an explicit formulation

$$x_n = f(n),$$

that is, the n-th number is a function of its 'key', n. Adding a block key into the equation

$$x_n^{(k)} = f_k(n)$$

allows for parallelism over processes indexed by k. See [170].

19.3.5 Mersenne twister

The *Mersenne twister* random number generator has been adapted to allow for parallel streams of uncorrelated numbers [147]. Here the process ID is encoded into the generator.

Chapter 20

Graph theory

Graph theory is the branch of mathematics that studies pairwise relations between objects. Graphs both appear as tools for analyzing issues in HPC, and as objects of study themselves. This appendix introduces the basic concepts and some relevant theory.

20.1 Definitions

A graph consists of a set of objects, and set of relations between them. The objects, called the *nodes* or *vertices* of the graph, usually form a finite set, so we usually identify them with consecutive integers $1 \ldots n$ or $0 \ldots n-1$. The relation that holds between nodes is described by the *edges* of the graph: if i and j are related, we say that (i, j) is an edge of the graph. This relation does not need to be symmetric, take for instance the 'less than' relation.

Formally, then, a graph is a tuple $G = \langle V, E \rangle$ where $V = \{1, \ldots n\}$ for some n, and $E \subset \{(i, j) : 1 \leq i, j \leq n,\ i \neq j\}$.

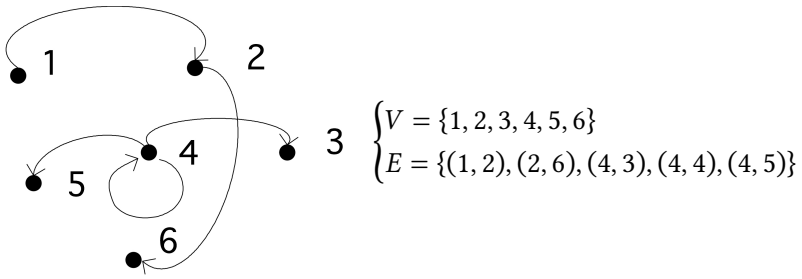


Figure 20.1: A simple graph.

A graph is called an *undirected graph* if $(i, j) \in E \Leftrightarrow (j, i) \in E$. The alternative is a *directed graph*, where we indicate an edge (i, j) with an arrow from i to j.

Two concepts that often appear in graph theory are the degree and the diameter of a graph.

Definition 3 *The* degree *denotes the maximum number of nodes that are connected to any node:*

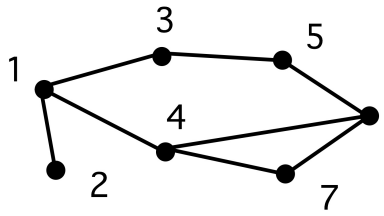
$$d(G) \equiv \max_i |\{j : j \neq i \wedge (i, j) \in E\}| .$$

Definition 4 *The* diameter *of a graph is the length of the longest shortest path in the graph, where a* path *is defined as a set of vertices $v_1, \ldots, v_{k+1}$ such that $v_i \neq v_j$ for all $i \neq j$ and*

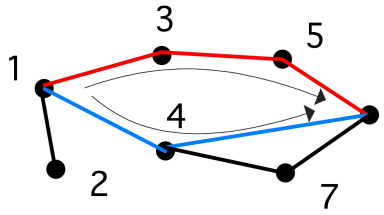
$$\forall_{1 \leq i \leq k} : (v_i, v_{i+1}) \in E.$$

The length of this path is k.

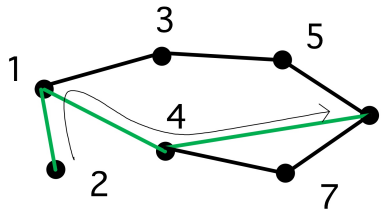
The concept of diameter is illustrated in figure 20.2.



A graph



Two paths from 1 to 6; $\langle 1, 4, 6 \rangle$ is the shorter



The longest shortest path of this graph

Figure 20.2: Shortest paths.

A path where all nodes are disjoint except for $v_1 = v_{k+1}$ is called a *cycle*.

Sometimes we are only interested in the mere existence of an edge (i, j), at other times we attach a value or 'weight' w_{ij} to that edge. A graph with weighted edges is called a *weighted graph*. Such a graph can be represented as a tuple $G = \langle V, E, W \rangle$ where E and W have the same cardinality. Conversely, an *unweighted graph* has all weights the same value, in which case we omit mention of weights.

20.2 Common types of graphs

20.2.1 Directed Acyclic Graphs

A graph that does not have cycles is called an *acyclic graph* A special case of this type of graph is the *Directed Acyclic Graph (DAG)*. This type of graph can for instance be used to model dependencies between tasks: if there is an edge from i to j, it means that task i has to be done before task j.

20.2.2 Trees

One special case of DAGs is the *tree graph*, or for short a *tree*: here any node can have multiple outgoing edges, but only one incoming edge. Nodes with no outgoing edges are *leaf nodes*; a node with no incoming edges is called a root, and all other nodes are called *interior nodes*.

Exercise 20.1. Can a tree have more than one root?

20.2.3 Separable graphs

A *separable graph* is a graph consisting of two sets of nodes, where all connections are inside either of the sets. Such graph can obviously be processed in parallel, so several parallelization algorithms are concerned with transforming a graph to a separable one. If a graph can be written as $V = V_1 + V_2 + S$, where nodes in V_1, V_2 are only connected to S, but not to the other set, we call S a *separator*.

20.2.4 Bipartite graphs

If a graph can be partitioned into two sets of nodes, where edges only run from the one set to the other, but not inside a set, we call this a *bipartite graph*.

20.3 Graph colouring and independent sets

We can assign labels to the nodes of a graph, which is equivalent to partitioning the set of nodes into disjoint subsets. One type of labeling that is of interest is *graph colouring*: here the labels (or 'colours') are chosen so that, if nodes i and j have the same colour, there is no edge connecting them: $(i, j) \notin E$.

There is a trivial colouring of a graph, where each node has its own colour. More interestingly, the minimum number of colours with which you can colour a graph is called the *colour number* of the graph.

Exercise 20.2. Show that, if a graph has degree d, the colour number is at most $d + 1$.

A famous graph colouring problem is the 'four colour theorem': if a graph depicts countries on a two-dimensional map (a so-called 'planar' graph), then the colour number is at most four. In general, finding the colour number is hard (in fact, NP-hard).

The colour sets of a graph colouring are also called *independent sets*, since within each colour no node is connected to a node of the same colour.

There is a trivial way of finding independent sets: declare each node to have its own unique colour. On the other hand, finding the 'best' division in independent sets, for instance through finding the colour number of the graph, is difficult. However, often it is enough to find a reasonable partitioning of the nodes into independent sets, for instance in constructing paralell ILU preconditioners; section 7.7.5. The following algorithm does that [115, 144]:

- Give each node a unique random number.
- Now find the set of nodes that have a higher number than all of their neighbors; call this the first independent set.

- Remove this set from the graph, and find again the nodes with a higher number than all their neighbors; this will be the second set.
- Repeat this procedure until all nodes are in an independent set.

Exercise 20.3. Convince yourself that the sets found this way are indeed independent.

20.4 Graph algorithms

In this section we only briefly touch on graph algorithms; a full discussion can be found in section 10.1.

- Distance algorithms. In applications such as traffic routing it is of interest to know the shortest distance from a given node to all others: the *single source shortest path* problem; or the shortest distance for any pair: the *all-pairs shortest path* problem.
- Connectivity algorithms. In social networks it is of interest to know if two people are connected at all, if the graph can be split up into unconnected subgraphs, or whether some person is the crucial bridge between two groups.

20.5 Graphs and matrices

A graph can be rendered in a number of ways. You could of course just list nodes and edges, but little insight can be derived that way. Simple graphs can be visualized by drawing vertices and edges, but for large graphs this becomes unwieldy. Another option is to construct the *adjacency matrix* of the graph. For a graph $G = \langle V, E \rangle$, the adjacency matrix M (with a size n equal to the number of vertices $|V|$) is defined by

$$M_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Conversely, if you have a matrix, especially a *sparse matrix*, you can construct its *adjacency graph*. This is illustrated in figure 20.3 for both a dense and a sparse matrix. In this example, the matrices are structurally

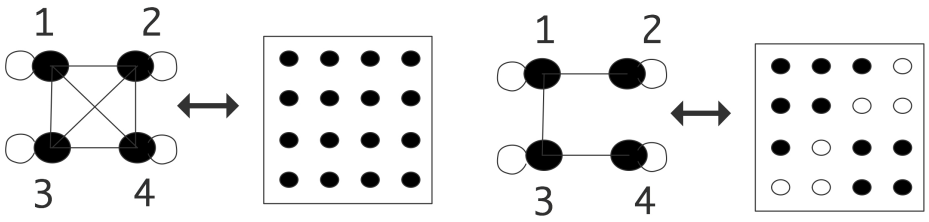


Figure 20.3: A dense and a sparse matrix, both with their adjacency graph.

symmetric, so we use lines instead of arrows in the graphs. There is an edge on each vertex corresponding to the diagonal element; this edge will often be left out of illustrations.

For graphs with edge weights, we set the elements of the adjacency matrix to the weights:

$$M_{ij} = \begin{cases} w_{ij} & (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

If a matrix has no zero elements, its adjacency graph has an edge between each pair of vertices. Such a graph is called a *clique*. If the graph is undirected, the adjacency matrix is symmetric, and conversely, if a matrix is *structurally symmetric*, its adjacency graph is undirected.

20.5.1 Permutation

Graphs are often used to indicate relations between objects in the real world. One example would be 'friend-of' relations in Facebook. In such cases, the nodes in a graph do not have a natural numbering: they are identified by a name, and any numbering is artificial. Thus, we could wonder which graph properties remain invariant, and which ones change, if we apply a different numbering.

Renumbering a set of objects can be modeled algebraically by multiplying the adjacency matrix by a *permutation matrix*.

Definition 5 *A permutation matrix is a square matrix where each row and column has exactly one element equal to one; all other elements are zero.*

Exercise 20.4. Let a set of N objects $x_1, \ldots, x_N$ be given. What is the permutation matrix that orders them as $x_1, x_3, \ldots, x_2, x_4, \ldots$? That is, find the matrix P such that

$$\begin{pmatrix} x_1 \\ x_3 \\ \vdots \\ x_2 \\ x_4 \\ \vdots \end{pmatrix} = P \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix}$$

Exercise 20.5. Show that the eigenvalues of a matrix are invariant under permutation.

20.5.2 Irreducibility

As an example of a graph concept that has an easy interpretation in the adjacency matrix, consider reducibility.

Definition 6 *A (directed) graph is called* irreducible *if for every pair i, j of nodes there is a path from i to j and from j to i. A graph is reducible if it is not irreducible.*

Exercise 20.6. Let A be a matrix

$$A = \begin{pmatrix} B & C \\ \emptyset & D \end{pmatrix} \tag{20.1}$$

where B and D are square matrices. Prove the reducibility of the graph of which this is the adjacency matrix.

The matrix in equation 20.1 is block upper triangular matrix. This means that solving a system $Ax = b$ is solved in two steps, each of size $N/2$, if N is the size of A.

Exercise 20.7. Show that this makes the arithmetic complexity of solving $Ax = b$ lower than for a general $N \times N$ matrix.

If we permute a graph, its reducibility or irreducibility is not changed. However, it may now no longer be apparent from looking at the adjacency matrix.

20.5.3 Graph closure

Here is another example of how adjacency matrices can simplify reasoning about graphs.

Definition 7 *Let $G = \langle V, E \rangle$ be an undirected graph, then we define the closure of G as $G' = \langle V, E' \rangle$ be the graph with the same vertices, but with vertices defined by*

$$(i, j) \in E' \Leftrightarrow \exists_k : (i, k) \in E \wedge (k, j) \in E.$$

As a small example:



Exercise 20.8. If M is the adjacency matrix of G, show that M^2 is the adjacency matrix of G', where we use boolean multiplication on the elements: $1 \cdot 1 = 1$, $1 + 1 = 1$.

20.6 Spectral graph theory

With a graph G[1] and its adjacency matrix A_G, we can define a *stochastic matrix* or *Markov matrix* by scaling A_G to have unit row sums:

$$W_G = D_G^{-1} A_G \qquad \text{where } (D_G)_{ii} = \deg(i).$$

To see how we interpret this, let's look at a simple example. Let's take an unweighted graph with an adjacency matrix

$$A_G = \begin{pmatrix} 1 & & 1 & 1 \\ & & 1 & 1 \\ 1 & & 1 & 1 \\ & 1 & 1 & \end{pmatrix}$$

1. This section owes much to Dan Spielman's course on spectral graph theory http://www.cs.yale.edu/homes/spielman/561/.

and look at the second row, which says that there are edges $(2,3)$ and $(2,4)$. This means that if you are on node 2, you can go to nodes 3 and 4. Scaling this matrix we get

$$W_G = \begin{pmatrix} 1/3 & & 1/3 & 1/3 \\ & & 1/2 & 1/2 \\ 1/3 & & 1/3 & 1/3 \\ & 1/2 & 1/2 & \end{pmatrix}$$

and now the second row says that from node 2 you can get with equal probability to nodes 3 and 4. You can also derive this statement mathematically:

$$\begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix} W_G = \begin{pmatrix} 0 & 0 & 1/2 & 1/2 \end{pmatrix}$$

It is simple to extrapolate that: if p is a vector where the i-th component gives the probability of being in node i, then $(p^t W_G)_i$ is the probability of being in node i if you take one more step along a graph edge.

Exercise 20.9. Prove that $p^t W_G$ is indeed a vector of probabilities. Hint: you can express that p is a probability vector as $p^t e = e$, where e is the vector of all ones.

20.6.1 The graph Laplacian

Another matrix to associate with a graph is the *graph Laplacian*

$$L_G = D_G - A_G.$$

This matrix has zero rowsums and positive diagonal entries, so by the *Gershgorin theorem* (section 14.5 all its eigenvalues are in the complex right half plane.

Exercise 20.10. Show that the vector of all ones is an eigenvector with eigenvalue 1.

This Laplacian matrix gives us a quadratic form:

$$x^t L_G x = \sum_{(i,j)\in E} (x_i - x_j)^2.$$

20.6.2 Domain decomposition through Laplacian matrices

There are various interesting theorems connected with the graph adjacency and Laplacian matrix. These have a very practical application to *domain decomposition*.

We get our inspiration of elliptic PDEs.

Connected with the *Laplace equation* $-\Delta u = f$ is an operator $Lu = -\Delta u$. On the unit interval $[0,1]$ the eigenfunctions of this operator, that is, the functions for which $Lu = \lambda u$, are $u_n(x) = \sin n\pi x$ for $n > 0$. These have the property that $u_n(x)$ has $n-1$ zeros in the interior of the interval, and they divide the interval in n connected regions where the function is positive of negative. Thus, if you wanted to divide a domain Ω over p processors, you could consider the p-th eigenfunction of the Laplacian on Ω, and find the connected regions where it is positive or negative.

This statement about PDE has a graph equivalent in two versions of *Fiedler's theorem*. (We will not give any proofs in this section; see [178].)

Theorem 7 *Let G be a weighted path graph on n vertices, let L_P have eigenvalues $0 = \lambda_1 < \lambda_2 \leq ... \leq \lambda_n$, and let v_k be an eigenvector of λ_k. Then v_k changes sign $k-1$ times.*

The second theorem is more useful [62]:

Theorem 8 *Let $G = (V, E, w)$ be a weighted connected graph, and let L_G be its Laplacian matrix. Let $0 = \lambda_1 < \lambda_2 \leq \cdots \leq \lambda_n$ be the eigenvalues of L_G and let $v_1, \ldots, v_n$ be the corresponding eigenvectors. For any $k \geq 2$, let $W_k = \{i \in V : v_k(i) \geq 0\}$. Then, the graph induced by G on W_k has at most $k-1$ connected components.*

The important consequence of this is that the eigenvector to the first nontrivial eigenvalue can be used to partition the graph in two connected piecesone of nodes where the eigenvector is positive, and one where the eigenvector is negative. This eigenvector is known as the *Fiedler vector*. The adjacency matrix is nonnegative, and there is an extensive theory for this type of matrix [12]; see the Perron-Frobenius theorem in section 14.4.

In general there are no guarantees for how good a decomposition this is, measured by the ratio of the numbers of edges, but in practice it can be shown that the behaviour is pretty good [179].

20.6.3 Cheeger's inequality

Above we remarked that the first non-trivial eigenvalue of the graph Laplacian has a relation to partitioning a graph in two parts. The *Cheeger's constant* and *Cheeger's inequality* relate this eigenvalue to a certain quality measure of partitionings.

Let V be the set of vertices and $S \subset V$, then Cheeger's constant of a graph is defined as

$$C = \min_S \frac{e(S, V-S)}{\min \mathrm{vol}(S), \mathrm{vol}(V-S)}$$

where $e(S, V-S)$ denotes the number of edges connecting S to $V-S$, and the volume of a set of nodes is defined as

$$\mathrm{vol}(S) = \sum_{e \in S} d(e).$$

Cheeger's inequality then states

$$2C \geq \lambda \geq \frac{C^2}{2}$$

where λ is the first nontrivial eigenvalue of the graph Laplacian.

Chapter 21

Automata theory

Automata are mathematical abstractions of machines. There is an extensive theory of automata; here we will only touch on the basic concepts. Let us start with a simple example.

21.1 Finite State Automata

A *Finite State Automaton (FSA)* is a very simple machine, something on the order of a vending machine that will dispense a candy bar when a quarter has been inserted. There are four actions possible with a vending machine: insert a quarter, press 'coin return' to ask for any inserted money back, open the window to take the candy bar, and close the window again. Whether an action is possible (especially the third) depends on the *state* the machine is in. There are three states: the begin state, the state where the quarter has been inserted and the window unlocked (let us call this 'ready to dispense'), and the state where the window is open (which we will call 'dispensing').

In certain states, certain actions are not possible. For instance, in the beginning state the window can not be opened.

The mathematical description of this vending machine consists of 1. the list of states, 2. a table of how the possible actions make the machine go from one state to another. However, rather than writing down the table, a graphical representation is usually more insightful.

21.2 General discussion

From the vending machine example, you saw an important characteristic of an automaton: it allows certain actions, but only in some circumstances, and in the end it has a state corresponding to 'success', which is only reached if certain sequences of actions are taken.

The formal description of this process is as follows: we call the individual actions an 'alphabet', and a sequence of actions a 'word' from based on that alphabet. The 'success' result of a machine then corresponds to a verdict that a certain word belong to the 'language' that is accepted by the automaton. Thus there is a correspondence between automata theory and language theory.

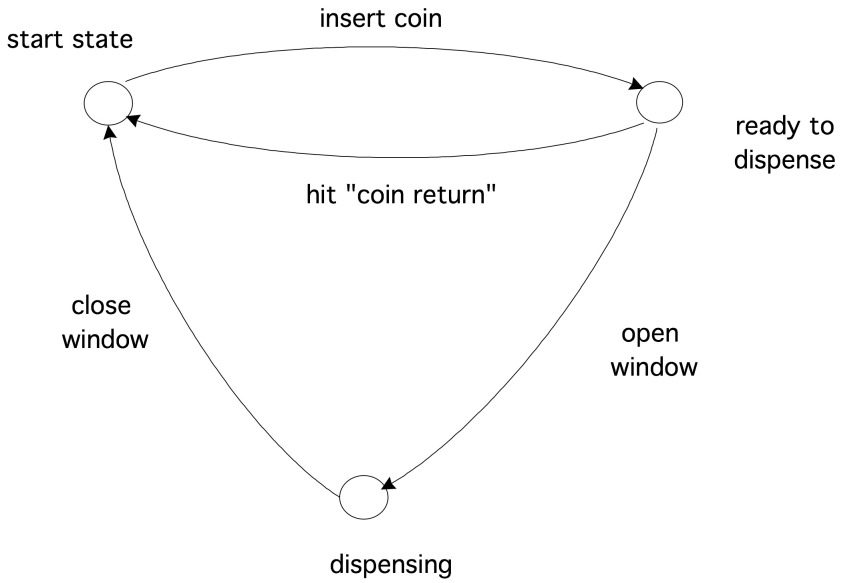


Figure 21.1: A simple real-life automaton.

Exercise 21.1. Consider the alpha $\{a, b\}$, that is, the alphabet with only the letters a, b, and consider the language $\{a^m b^n : m, n > 0\}$, that is the words that consist of one or more as followed by one or more bs. Draw the automaton that accepts this language.

What makes the FSA the simplest type is that it has no memory. Most vending machines do not complain if you put in more than one quarter: they have no memory beyond 'a quarter has been inserted'. A more complicated machine would count how many quarters you inserted, and then allow you to open that many windows to different candy bars. In the above formal way of describing, that machine would accept the language $\{q^n w^n : n \geq 0\}$, that is, the sequences where you deposit as many quarters ('q') as you open windows ('w'). This language is an example of a so-called *context-free language*; the language of the original vending machine is a *regular language*.

These two language types belong to the four level *Chomsky hierarchy* of languages. The famous *Turing machine*, which recognizes the *recursively enumerable language* type, is on the top level of the hierarchy. The missing step has the *context-sensitive language* type, which is recognized by a *linear bounded automaton*.

Chapter 22

Parallel Prefix

For operations to be executable in parallel they need to be independent. That makes recurrences problematic to evaluate in parallel. Recurrences occur in obvious places such as solving a triangular system of equations (section 5.3.4), but they can also appear in sorting and many other operations.

In this appendix we look at *parallel prefix* operations: the parallel execution of an operation that is defined by a recurrence involving an associative operator. (See also section 7.10.2 for the 'recursive doubling' approach to parallelizing recurrences.) Computing the sum of an array of elements is an example of this type of operation (disregarding the *non-associativity* for the moment). Let $\pi(x,y)$ be the binary sum operator:

$$\pi(x,y) \equiv x+y,$$

then we define the prefix sum of $n \geq 2$ terms as

$$\Pi(x_1,\ldots,x_n) = \begin{cases} \pi(x_1,x_2) & \text{if } n=2 \\ \pi\big(\Pi(x_1,\ldots,x_{n-1}),x_n\big) & \text{otherwise} \end{cases}$$

As a non-obvious example of a prefix operation, we could count the number of elements of an array that have a certain property.

Exercise 22.1. Let $p(\cdot)$ be a predicate, $p(x)=1$ if it holds for x and 0 otherwise. Define a binary operator $\pi(x,y)$ so that its reduction over an array of numbers yields the number of elements for which p is true.

So let us now assume the existence of an associative operator $\oplus$, an array of values $x_1,\ldots,x_n$. Then we define the prefix problem as the computation of $X_1,\ldots,X_n$, where

$$\begin{cases} X_1 = x_1 \\ X_k = \oplus_{i\leq k} x_i \end{cases}$$

22.1 Parallel prefix

The key to parallelizing the prefix problem is the realization that we can compute partial reductions in parallel:

$$x_1 \oplus x_2, \quad x_3 \oplus x_4, \ldots$$

are all independent. Furthermore, partial reductions of these reductions,

$$(x_1 \oplus x_2) \oplus (x_3 \oplus x_4), \quad \ldots$$

are also independent. We use the notation

$$X_{i,j} = x_i \oplus \cdots \oplus x_j$$

for these partial reductions.

You have seen this before in section 2.1: an array of n numbers can be reduced in $\lceil \log_2 n \rceil$ steps. What is missing in the summation algorithm to make this a full prefix operation is computation of all intermediate values.

Observing that, for instance, $X_3 = (x_1 \oplus x_2) \oplus x_3 = X_2 \oplus x_3$, you can now imagine the whole process; see figure 22.1 for the case of 8 elements. To compute, say, X_{13}, you express $13 = 8 + 4 + 1$ and compute

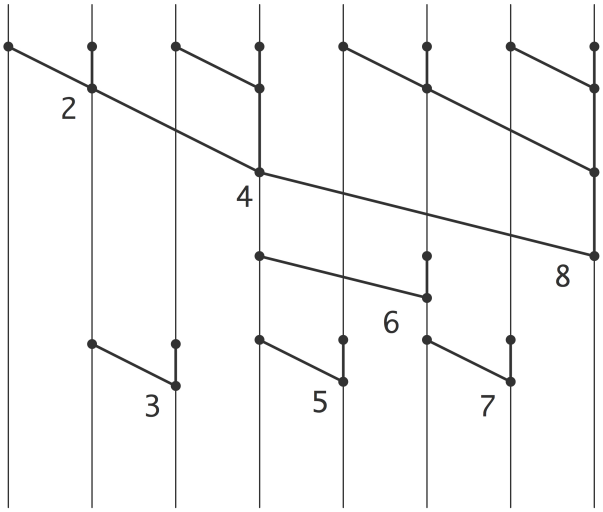


Figure 22.1: A prefix operation applied to 8 elements.

$X_{13} = X_8 \oplus X_{9,12} \oplus x_{13}$.

In this figure, operations over the same 'distance' have been vertically aligned corresponding to a SIMD type execution. If the execution proceeds with a task graph, some steps can be executed earlier than the figure suggests; for instance X_3 can be computed simultaneously with X_6.

Regardless the arrangement of the computational steps, it is not hard to see that the whole prefix calculation can be done in $2 \log_2 n$ steps: $\log_2 n$ steps for computing the final reduction X_n, then another $\log_2 n$ steps for filling in the intermediate values.

22.2 Sparse matrix vector product as parallel prefix

It has been observed that the sparse matrix vector product can be considered a prefix operation; see [16]. The reasoning here is we first compute all $y_{ij} \equiv a_{ij}x_j$, and subsequently compute the sums $y_i = \sum_j y_{ij}$ with

a prefix operation.

A prefix sum as explained above does not compute the right result. The first couple of y_{ij} terms do indeed sum to y_1, but then continuing the prefix sum gives $y_1 + y_2$, instead of y_2. The trick to making this work is to consider two-component quantities $\langle y_{ij}, s_{ij} \rangle$, where

$$s_{ij} = \begin{cases} 1 & \text{if } j \text{ is the first nonzero index in row } i \\ 0 & \text{otherwise} \end{cases}$$

Now we can define prefix sums that are 'reset' every time $s_{ij} = 1$.

22.3 Horner's rule

Horner's rule for evaluating a polynomial is an example of a simple recurrence:

$$y = c_0 x^n + \cdots + c_n x^0 \equiv \begin{cases} t_0 \leftarrow c_0 \\ t_i \leftarrow t_{i-1} \cdot x + c_i \quad i = 1, \ldots, n \\ y = t_n \end{cases} \tag{22.1}$$

or, written more explicitly

$$y = (((c_0 \cdot x + c_1) \cdot x + c_2) \cdots) .$$

Like many other recurrences, this seemingly sequential operation can be parallelized:

$$\begin{array}{cccc} c_0 x + c_1 & c_2 x + c_3 & c_4 x + c_5 & c_6 x + c_7 \\ \cdot \times x^2 + \cdot & & \cdot \times x^2 + \cdot & \\ & \cdot \times x^4 + \cdot & & \end{array}$$

However, we see here that some cleverness is needed: we need x, x^2, x^4 etc. to multiply subresults.

Interpreting Horner's rule as a prefix scheme fails: the 'horner operator' $h_x(a, b) = ax + b$ is easily seen not to be associative. From the above treewise calculation we see that we need to carry and update the x, rather than attaching it to the operator.

A little experimenting shows that

$$h\left(\begin{bmatrix} a \\ x \end{bmatrix}, \begin{bmatrix} b \\ y \end{bmatrix}\right) \equiv \begin{bmatrix} ay + b \\ xy \end{bmatrix}$$

serves our purposes:

$$h\left(\begin{bmatrix} a \\ x \end{bmatrix}, \begin{bmatrix} b \\ y \end{bmatrix}, \begin{bmatrix} c \\ z \end{bmatrix}\right) = \left\{ \begin{array}{rcl} h\left(h\left(\begin{bmatrix} a \\ x \end{bmatrix}, \begin{bmatrix} b \\ y \end{bmatrix}\right), \begin{bmatrix} c \\ z \end{bmatrix}\right) & = & h\left(\begin{bmatrix} ay + b \\ xy \end{bmatrix}, \begin{bmatrix} c \\ z \end{bmatrix}\right) \\ h\left(\begin{bmatrix} a \\ x \end{bmatrix}, h\left(\begin{bmatrix} b \\ y \end{bmatrix}, \begin{bmatrix} c \\ z \end{bmatrix}\right)\right) & = & h\left(\begin{bmatrix} a \\ x \end{bmatrix}, \begin{bmatrix} bz + c \\ yz \end{bmatrix}\right) \end{array} \right\} = \begin{bmatrix} ayz + bz + c \\ xyz \end{bmatrix}$$

With this we can realize Horner's rule as

$$h\left(\begin{bmatrix} c_0 \\ x \end{bmatrix}, \begin{bmatrix} c_1 \\ x \end{bmatrix}, \ldots, \begin{bmatrix} c_n \\ x \end{bmatrix}\right)$$

As an aside, this particular form of the 'horner operator' corresponds to the 'rho' operator in the programming language *APL*, which is normally phrased as evaluation in a number system with varying radix.

PART IV

PROJECTS, CODES

Chapter 23

Class projects

Here are some suggestions for end-of-semester projects that feature a combination of coding and analysis.

Chapter 24

Teaching guide

The material in this book is far more than will fit a one-semester course. Here are a couple of strategies for how to teach it as a course, or incorporate it in an other course.

24.1 Standalone course

24.2 Addendum to parallel programming class

Suggested teachings and exercises:

Parallel analysis: 2.12, 2.13, 2.16.

Architecture: 2.32, 2.33, 2.35, 2.37.

Complexity: 2.38

24.3 Tutorials

Chapter 25

Cache simulation and analysis

In this project you will build a cache simulator and analyze the cache hit/miss behaviour of code, either real or simulated.

25.1 Cache simulation

A simulated cache is a simple data structure that records for each cache address what memory address it contains, and how long the data has been present. Design this data structure and write the access routines. (Use of an object oriented language is recommended.)

Write your code so that your cache can have various levels of associativity, and different replacement policies.

For simplicity, do not distinguish between read and write access. Therefore, the execution of a program becomes a stream of

```
cache.access_address( 123456 );
cache.access_address( 70543 );
cache.access_address( 12338383 );
.....
```

calls where the argument is the memory address. Your code will record whether the request can be satisfied from cache, or whether the data needs to be loaded from memory.

25.2 Code simulation

Find some example codes, for instance from a scientific project you are involved in, and translate the code to a sequence of memory address.

You can also simulate codes by generating a stream of access instructions as above:

- Some access will be to random locations, corresponding to use of scalar variables;
- At other times access will be to regularly space addresses, corresponding to the use of an array;

- Array operations with indirect addressing will cause prolonged episodes of random address access.

Try out various mixes of the above instruction types. For the array operations, try smaller and larger arrays with various degrees of reuse.

Does your simulated code behave like a real code?

25.3 Investigation

First implement a single cache level and investigate the behaviour of cache hits and misses. Explore different associativity amounts and different replacement policies.

25.4 Analysis

Do a statistical analysis of the cache hit/miss behaviour. You can start with [167][1]. Hartstein [96] found a power law behaviour. Are you finding the same?

1. Strictly speaking that paper is about page swapping out of virtual memory (section 1.3.9.2), but everything translates to cacheline swapping out of main memory.

Chapter 26

Evaluation of Bulk Synchronous Programming

In this project you're asked to analyze the Bulk Synchronous Parallel (BSP) model, both abstractly and by simulation.

26.1 Discussion

For your assignment you need to investigate Bulk Synchronous Programming. Read the wikipedia article http://en.wikipedia.org/wiki/Bulk_synchronous_parallel and section 2.6.8 in the book.

Consider the following questions.

1. Discuss the relation between the complexity model of BSP and the α, β, γ model you learned in section 7.1.
2. BSP uses barrier synchronisations. Are these necesary in practice? Consider separately two different classes of algorithms, namely PDE solving, and large scale graph computations.

26.2 Simulation

In the defining paper on BSP, Valiant [185] advocates making more tasks than processors, and distributing them randomly. Program a simulation and test if this strategy solves the load balance problem. Discuss the random placement strategy in the context of various algorithms.

Chapter 27

Heat equation

In this project you will combine some of the theory and practical skills you learned in class to solve a real world problem. In addition to writing code that solves the program, you will use the following software practices:

- use source code control,
- give the program checkpoint/restart capability, and
- use basic profiling and visualization of the results.

The Heat Equation (see section 4.3) is given by

$$\frac{\partial T(x,t)}{\partial t} = \begin{cases} \alpha \frac{\partial^2 T(x,y)}{\partial x^2} + q(x,t) & \text{1D} \\ \alpha \frac{\partial^2 T(x,y)}{\partial x^2} + \alpha \frac{\partial^2 T(x,y)}{\partial y^2} + q(x,t) & \text{2D} \\ \ldots & \text{3D} \end{cases}$$

where $t \geq 0$ and $x \in [0,1]$, subject to boundary conditions

$$T(x,0) = T_0(x), \quad \text{for } x \in [0,1],$$

and similar in higher dimensions, and

$$T(0,t) = T_a(t),\ T(1,t) = T_b(t), \quad \text{for } t > 0.$$

You will solve this problem using the explicit and implicit Euler methods.

27.1 Software

As PDEs go, this is a fairly simple one. In particular the regular structure makes it easy to code this project using regular arrays to store the data. However, you can also write your software using the PETSc library. In that case, use the `MatMult` routine for matrix-vector multiplication and `KSPSolve` for linear system solution. Exception: code the Euler methods yourself.

Be sure to use a Makefile for building your project (tutorial *Tutorials book, section 3*).

Add your source files, Makefile, and job scripts to a git repository (tutorial *Tutorials book, section 5*); do not add binaries or output files. Make sure that there is a README file with instructions on how to build and run your code.

Implement a checkpoint/restart facility by writing vector data, size of the time step, and other necessary items, to an `hdf5` file (tutorial *Tutorials book, section 7*). Your program should be able to read this file and resume execution.

27.2 Tests

Do the following tests on a single core.

Method stability

Run your program for the 1D case with

$$\begin{cases} q = \sin \ell \pi x \\ T_0(x) = e^x \\ T_a(t) = T_b(t) = 0 \\ \alpha = 1 \end{cases}$$

Take a space discretization at least $h = 10^{-2}$ but especially in parallel do not be afraid to try large problem sizes. Try various time steps and show that the explicit method can diverge. What is the maximum time step for which it is stable?

For the implicit method, at first use a direct method to solve the system. This corresponds to PETSc options `KSPPREONLY` and `PCLU` (see section 5.5.11).

Now use an iterative method (for instance `KSPCG` and `PCJACOBI`); is the method still stable? Explore using a low convergence tolerance and large time steps.

Since the forcing function q and the boundary conditions have no time dependence, the solution $u(\cdot, t)$ will converge to a *steady state* solution $u_\infty(x)$ as $t \to \infty$. What is the influence of the time step on the speed with which implicit method converges to this steady state?

Hint: the steady state is described by $u_t \equiv 0$. Substitute this in the PDE. Can you find explicitly what the steady state is?

Run these tests with various values for ℓ.

Timing

If you run your code with the commandline option `-log_summary`, you will get a table of timings of the various PETSc routines. Use that to do the following timing experiments. Make sure you use a version of PETSc that was *not* compiled with debug mode.

Construct your coefficient matrix as a dense matrix, rather than sparse. Report on the difference in total memory, and the runtime and flop rate of doing one time step. Do this for both the explicit and implicit method and explain the results.

With a sparse coefficient matrix, report on the timing of a single time step. Discuss the respective flop counts and the resulting performance.

Restart

Implement a restart facility: every 10 iterations write out the values of the current iterate, together with values of Δx, Δt, and ℓ. Add a flag `-restart` to your program that causes it to read the restart file and resume execution, reading all parameters from the restart file.

Run your program for 25 iterations, and restart, causing it to run again from iteration 20. Check that the values in iterations 20 ... 25 match.

27.3 Parallelism

Do the following tests to determine the parallel scaling of your code.

At first test the explicit method, which should be perfectly parallel. Report on actual speedup attained. Try larger and smaller problem sizes and report on the influence of the problem size.

The above settings for the implicit method (`KSPPREONLY` and `PCLU`) lead to a runtime error. One way out is to let the system be solved by an iterative method. Read the PETSc manual and web pages to find out some choices of iterative method and preconditioner and try them. Report on their efficacy.

27.4 Comparison of solvers

In the implicit timestepping method you need to solve linear systems. In the 2D (or 3D) case it can make a big difference whether you use a direct solver or an iterative one.

Set up your code to run with $q \equiv 0$ and zero boundary conditions everywhere; start with a nonzero initial solution. You should now get convergence to a zero steady state, thus the norm of the current solution is the norm of the iterate.

Now do the following comparison; take several values for the time step.

Direct solver

If your PETSc installation includes direct solvers such as *MUMPS*, you can invoke them with

```
myprog -pc_type lu -ksp_type preonly \
    -pc_factor_mat_solver_package mumps
```

Run your code with a direct solver, both sequentially and in parallel, and record how long it takes for the error to get down to 10^{-6}.

Iterative Solver

Use an iterative solver, for instance `KSPCG` and `KSPBCGS`. Experiment with the convergence tolerance: how many timesteps does it take to get a 10^{-6} error if you set the iterative method tolerance to 10^{-12}, how much if you take a lesser tolerance?

Compare timings between direct and iterative method.

27.5 Reporting

Write your report using LaTeX (tutorial *Tutorials book, section 15*). Use both tables and graphs to report numerical results. Use `gnuplot` (tutorial *Tutorials book, section 9*) or a related utility for graphs.

Chapter 28

The memory wall

In this project you explore the ramifications of the *memory wall*; see section 1.3. This project involves literature search, but little programming.

28.1 Background

Wulf and McKee [198] observed trends in the average latency in processors. Let t_m be the latency from memory, t_c the latency from cache, and p the probability of a cache hit, then the average latency is

$$t_{\text{avg}} = pt_c + (1-p)t_m.$$

As the gap between processor speed and memory speed increases, this latency will grow, unless one can drive p down, that is, reduce the number of cache misses or at least lessen their importance.

28.2 Assignment

Do a literature search and discuss the following topics.

- What have the trends been in processor speed and memory speed? What bearing does the introduction of multicore chips have on this balance?
- Section 1.3.5.5 discussed various types of cache misses. Some misses are more related to the algorithm and some more to the hardware. What strategies have hardware designers used to less then impact of cache misses?
- Compulsory cache misses seem unavoidable, since they are a property of the algorithm. However, if they can be hidden (see section 1.3.2 for 'latency hiding') their performance impact disappears. Research the topic of prefetch streams and their relation to latency hiding.
- How does the size of a cacheline interact with this behaviour?
- Discuss compiler techniques that lessen the incidence of cache misses.
- Can you find examples of algorithm options, that is, algorithms that compute the same result (not necessarily in the arithmetic sense; compare direct square root versus iterative approximation) but with different computational behaviour?

For all of these topics you are encouraged to write simulations.

Chapter 29

Codes

This section contains several simple codes that illustrate various issues relating to the performance of a single CPU. The explanations can be found in chapter 6.

29.1 Preliminaries

29.1.1 Hardware event counting

The codes in this chapter make calls to a library named PAPI for 'Performance Application Programming Interface' [23, 161]. This is a portable set of calls to query the hardware counters that are built into most processors. Since these counters are part of the processor hardware, they can measure detailed events such as cache misses without this measurement process disturbing the phenomenon it is supposed to observe.

While using hardware counters is fairly straightforward, the question of whether what they are reporting is what you actually meant to measure is another matter altogether. For instance, the presence of hardware prefetch streams (section 1.3.6) implies that data can be loaded into cache without this load being triggered by a cache miss. Thus, the counters may report numbers that seem off, or even impossible, under a naive interpretation.

29.1.2 Test setup

In the next sections you will see several codes that try to do precise timings on some phenomenon. This requires some care if you want to be sure you are actuallying timing what you're interested in, and not something else.

First of all, of course you perform your experiment multiple times to get a good average timing. If the individual timings are too far apart, you can decide to throw away outliers, or maybe rethink your experiment: there may be a variable that you haven't accounted for.

Let's assume that your timings are all within an acceptable spread; next you have to compensate for cache effects stemming from the fact that you are doing an experiment multiple times. If your problem sizes is small, the first timing will bring all data in cache, and the second run will find it there, leading to much

faster runtimes. You can solve this problem by, in between any pair of timings, touching an array that is larger than the cache size. This has the effect of flushing all experiment data from the cache.

On the other hand, if you want to time something that happens on data in cache, you want to be sure that your data is in cache, so you could for instance write to the array, bringing it in cache, prior to your experiment. This is sometimes referred to as *cache warming*, and a cache that contains the problem data is called a *hot cache*. Failure to warm the cache leads to an results where the first run takes appreciably longer than the rest.

29.1.3 Memory issues

For performance reasons it is often desirable to enforce *cacheline boundary alignment* of data. The easiest solution is to use *static allocation*, which puts the data on cacheline boundaries.

To put dynamically aligned data on cacheline boundaries, the following code does the tric:

```
double *a;
a = malloc( /* some number of bytes */ +8 );
if ( (int)a % 8 != 0 ) { /* it is not 8-byte aligned */
  a += 1; /* advance address by 8 bytes, then */
  /* either: */
  a = ( (a>>3) <<3 );
  /* or: */
  a = 8 * ( ( (int)a )/8 );
}
```

This code allocates a block of memory, and, if necessary, shifts it right to have a starting address that is a multiple of 8.

However, a better solution is to use *posix_memalign*:

```
int posix_memalign(
  void **memptr, size_t alignment, size_t size);
```

which allocates `size` bytes, aligned to a multiple of `alignment` bytes. For example:

```
double x;
posix_memalign( (void**)&x,64,N*sizeof(double) );
```

will allocate 64 doubles for `x`, and align the array to cacheline boundaries if there are 8 words per cacheline.

(If the *Intel compiler* complains about `posix_memalign` being declared implicitly, add the `-std=gnu99` flag to the compile line.)

29.2 Cache size

This code demonstrates the fact that operations are more efficient if data is found in L1 cache, than in L2, L3, or main memory. To make sure we do not measure any unintended data movement, we perform one iteration to bring data in the cache before we start the timers.

`size.c`

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */

#define PCHECK(e) \
  if (e!=PAPI_OK) \
    {printf("Problem in papi call, line %d\n",__LINE__); return 1;}
#define NEVENTS 3
#define NRUNS 200
#define L1WORDS 8096
#define L2WORDS 100000

int main(int argc, char **argv)
{
  int events[NEVENTS] =
    {
      PAPI_TOT_CYC,/* total cycles */
      PAPI_L1_DCM, /* stalls on L1 cache miss */
      PAPI_L2_DCM, /* stalls on L2 cache miss */
    };
  long_long values[NEVENTS];
  PAPI_event_info_t info, info1;
  const PAPI_hw_info_t *hwinfo = NULL;
  int retval,event_code, m,n, i,j,size, arraysize;
  const PAPI_substrate_info_t *s = NULL;
  double *array;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   retval = PAPI_library_init(PAPI_VER_CURRENT);
   if (retval != PAPI_VER_CURRENT)
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);
   {
     int i;
     for (i=0; i<NEVENTS; i++) {
       retval = PAPI_query_event(events[i]) ; PCHECK(retval);
     }
   }

   /* declare an array that is more than twice the L2 cache size */
   arraysize=2*L2WORDS;
   array = (double*) malloc(arraysize*sizeof(double));

   for (size=L1WORDS/4; size<arraysize; size+=L1WORDS/4) {
     printf("Run: data set size=%d\n",size);

     /* clear the cache by dragging the whole array through it */
     for (n=0; n<arraysize; n++) array[n] = 0.;
     /* now load the data in the highest cache level that fits */
     for (n=0; n<size; n++) array[n] = 0.;

     retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
     /* run the experiment */
     for (i=0; i<NRUNS; i++) {
       for (j=0; j<size; j++) array[j] = 2.3*array[j]+1.2;
```

```
        }
        retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
        printf("size=%d\nTot cycles: %d\n",size,values[0]);
        printf("cycles per array loc: %9.5f\n",size,values[0]/(1.*NRUNS*size));
        printf("L1 misses:\t%d\nfraction of L1 lines missed:\t%9.5f\n",
        values[1],values[1]/(size/8.));
        printf("L2 misses:\t%d\nfraction of L2 lines missed:\t%9.5f\n",
        values[2],values[2]/(size/8.));
        printf("\n");
      }
      free(array);

      return 0;
    }
```

We also measure performance by repeatedly traversing a linked list array. This is created as:

```
// hardware/allocation.cpp
std::iota(indices.begin(),indices.end(),0);
std::random_device r;
std::mt19937 g(r());
std::shuffle(indices.begin(), indices.end(), g);
auto data = thecache;
for (size_t i=0; i<indices.size(); i++)
  data[i] = indices[i];
```

and traversed as:

```
// hardware/allocation.cpp
auto data = thecache;
for (size_t i=0; i<n_accesses; i++) {
  res = data[res];
  /* ... */
}
```

29.3 Cachelines

This code illustrates the need for small strides in vector code. The main loop operates on a vector, progressing by a constant stride. As the stride increases, runtime will increase, since the number of cachelines transferred increases, and the bandwidth is the dominant cost of the computation.

There are some subtleties to this code: in order to prevent accidental reuse of data in cache, the computation is preceded by a loop that accesses at least twice as much data as will fit in cache. As a result, the array is guaranteed not to be in cache.

`line.c`

```
    #include "papi_test.h"
    extern int TESTS_QUIET;         /* Declared in test_utils.c */

    #define PCHECK(e) \
      if (e!=PAPI_OK) \
```

```
    {printf("Problem in papi call, line %d\n",__LINE__); return 1;}
#define NEVENTS 4
#define MAXN 10000
#define L1WORDS 8096
#define MAXSTRIDE 16

int main(int argc, char **argv)
{
  int events[NEVENTS] =
    {PAPI_L1_DCM,/* stalls on L1 cache miss */
     PAPI_TOT_CYC,/* total cycles */
     PAPI_L1_DCA, /* cache accesses */
     1073872914 /* L1 refills */};
  long_long values[NEVENTS];
  PAPI_event_info_t info, info1;
  const PAPI_hw_info_t *hwinfo = NULL;
  int retval,event_code, m,n, i,j,stride, arraysize;
  const PAPI_substrate_info_t *s = NULL;
  double *array;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   retval = PAPI_library_init(PAPI_VER_CURRENT);
   if (retval != PAPI_VER_CURRENT)
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);
   {
     int i;
     for (i=0; i<NEVENTS; i++) {
       retval = PAPI_query_event(events[i]) ; PCHECK(retval);
     }
   }

   /* declare an array that is more than twice the cache size */
   arraysize=2*L1WORDS*MAXSTRIDE;
   array = (double*) malloc(arraysize*sizeof(double));

   for (stride=1; stride<=MAXSTRIDE; stride++) {
     printf("Run: stride=%d\n",stride);
     /* clear the cache by dragging the whole array through it */
     for (n=0; n<arraysize; n++) array[n] = 0.;

     retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
     /* run the experiment */
     for (i=0,n=0; i<L1WORDS; i++,n+=stride) array[n] = 2.3*array[n]+1.2;
     retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
     printf("stride=%d\nTot cycles: %d\n",stride,values[1]);
     printf("L1 misses:\t%d\naccesses per miss:\t%9.5f\n",
     values[0],(1.*L1WORDS)/values[0]);
     printf("L1 refills:\t%d\naccesses per refill:\t%9.5f\n",
     values[3],(1.*L1WORDS)/values[3]);
     printf("L1 accesses:\t%d\naccesses per operation:\t%9.5f\n",
     values[2],(1.*L1WORDS)/values[2]);
     printf("\n");
   }
```

```
   free(array);

   return 0;
}
```

29.4 Cache associativity

This code illustrates the effects of cache associativity;see sections 1.3.5.11 and 6.5 for a detailed explanation. A number of vectors (dependent on the inner loop variable `i`) is traversed simultaneously. Their lengths are chosen to induce cache conflicts. If the number of vectors is low enough, cache associativity will resolve these conflicts; for higher values of `m` the runtime will quickly increase. By allocating the vectors with a larger size, the cache conflicts go away.

`assoc.c`

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */

#define PCHECK(e) if (e!=PAPI_OK) \
        {printf("Problem in papi call, line %d\n",__LINE__); \
         return 1;}
#define NEVENTS 2
#define MAXN 20000

/* we are assuming array storage in C row mode */
#if defined(SHIFT)
#define INDEX(i,j,m,n) (i)*(n+8)+(j)
#else
#define INDEX(i,j,m,n) (i)*(n)+(j)
#endif

int main(int argc, char **argv)
{
   int events[NEVENTS] = {PAPI_L1_DCM,PAPI_TOT_CYC};
   long_long values[NEVENTS];
   PAPI_event_info_t info, info1;
   const PAPI_hw_info_t *hwinfo = NULL;
   int retval,event_code, m,n, i,j;
   const PAPI_substrate_info_t *s = NULL;
   double *array;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   retval = PAPI_library_init(PAPI_VER_CURRENT);
   if (retval != PAPI_VER_CURRENT)
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);
   {
     int i;
     for (i=0; i<NEVENTS; i++) {
       retval = PAPI_query_event(events[i]); PCHECK(retval);
     }
   }
```

```
   /*
   if (argc<3) {
     printf("Usage: assoc m n\n"); return 1;
   } else {
     m = atoi(argv[1]); n = atoi(argv[2]);
   } printf("m,n = %d,%d\n",m,n);
   */

#if defined(SHIFT)
   array = (double*) malloc(13*(MAXN+8)*sizeof(double));
#else
   array = (double*) malloc(13*MAXN*sizeof(double));
#endif

   /* clear the array and bring in cache if possible */
   for (m=1; m<12; m++) {
     for (n=2048; n<MAXN; n=2*n) {
       printf("Run: %d,%d\n",m,n);
#if defined(SHIFT)
       printf("shifted\n");
#endif

   for (i=0; i<=m; i++)
     for (j=0; j<n; j++)
       array[INDEX(i,j,m+1,n)] = 0.;

   /* access the rows in a way to cause cache conflicts */
   retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
   for (j=0; j<n; j++)
     for (i=1; i<=m; i++)
       array[INDEX(0,j,m+1,n)] += array[INDEX(i,j,m+1,n)];
   retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
   printf("m,n=%d,%d\n#elements:\t%d\nTot cycles: %d\n",
   m,n,m*n,values[1]);
   printf("L1 misses:\t%d\nmisses per accumulation:\t%9.5f\n\n",
   values[0],values[0]/(1.*n));

     }
   }
   free(array);

   return 0;
}
```

29.5 TLB

This code illustrates the behaviour of a *TLB*; see sections 1.3.9.2 and 6.4.4 for a thorough explanation. A two-dimensional array is declared in column-major ordering (Fortran style). This means that striding through the data by varying the i coordinate will have a high likelihood of TLB hits, since all elements on a page are accessed consecutively. The number of TLB entries accessed equals the number of elements

divided by the page size. Striding through the array by the *j* coordinate will have each next element hitting a new page, so TLB misses will ensue when the number of columns is larger than the number of TLB entries.

tlb.c

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */

#define PCHECK(e) if (e!=PAPI_OK) \
        {printf("Problem in papi call, line %d\n",__LINE__); \
         return 1;}
#define NEVENTS 2
/* we are assuming array storage in Fortran column mode */
#define INDEX(i,j,m,n) i+j*m

double *array;

void clear_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void clear_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void do_operation_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

void do_operation_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

#define COL 1
#define ROW 2
int main(int argc, char **argv)
{
```

```
   int events[NEVENTS] = {PAPI_TLB_DM,PAPI_TOT_CYC};
   long_long values[NEVENTS];
   int retval,order=COL;
   PAPI_event_info_t info, info1;
   const PAPI_hw_info_t *hwinfo = NULL;
   int event_code;
   const PAPI_substrate_info_t *s = NULL;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   if (argc==2 && !strcmp(argv[1],"row")) {
     printf("wrong way\n"); order=ROW;
   } else printf("right way\n");

   retval = PAPI_library_init(PAPI_VER_CURRENT);
   if (retval != PAPI_VER_CURRENT)
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);

   {
     int i;
     for (i=0; i<NEVENTS; i++) {
       retval = PAPI_query_event(events[i]); PCHECK(retval);
     }
   }

#define M 1000
#define N 2000
   {
     int m,n;
     m = M;
     array = (double*) malloc(M*N*sizeof(double));
     for (n=10; n<N; n+=10) {
       if (order==COL)
  clear_right(m,n);
       else
  clear_wrong(m,n);
       retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
       if (order==COL)
  do_operation_right(m,n);
       else
  do_operation_wrong(m,n);
       retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
       printf("m,n=%d,%d\n#elements:\t%d\n",m,n,m*n);
       printf("Tot cycles: %d\nTLB misses:\t%d\nmisses per column:\t%9.5f\n\n",
       values[1],values[0],values[0]/(1.*n));
     }
     free(array);
   }

   return 0;
}
```

PART V

INDICES

Chapter 30

Index

Chapter 31

List of acronyms

BEM Boundary Element Method
BLAS Basic Linear Algebra Subprograms
CCS Compressed Column Storage
CG Conjugate Gradients
CRS Compressed Row Storage
DL Deep Learning
DRAM Dynamic Random-Access Memory
FEM Finite Element Method
FMA Fused Multiply-Add
FEM Finite Element Method
FFT Fast Fourier Transform
FSB Front-Side Bus
GMRES Generalized Minimum Residual
GPU Graphics Processing Unit
HPC High-Performance Computing
ILP Instruction Level Parallelism
LAN Local Area Network
ML Machine Learning
NUMA Non-Uniform Memory Access
OS Operating System
PGAS Partitioned Global Address Space
PRAM Parallel Random Access Machine
SAN Storage Area Network
SAS Software As a Service
SIMD Single Instruction Multiple Data
SIMT Single Instruction Multiple Thread
SMP Symmetric Multi Processing
SPMD Single Program Multiple Data
SPD symmetric positive definite
SRAM Static Random-Access Memory
SSE SIMD Streaming Extensions
TLB Translation Look-aside Buffer
UMA Uniform Memory Access
WAN Wide Area Network

Chapter 32

Bibliography

[1] Loyce M. Adams and Harry F. Jordan. Is SOR color-blind? *SIAM J. Sci. Stat. Comput.*, 7:490–506, 1986. [Cited on page 315.]

[2] Sarita V. Adve and Hans-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53:90–101. [Cited on page 91.]

[3] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Computing Conference*, volume 30, pages 483–485, 1967. [Cited on page 65.]

[4] O. Axelsson and A.V. Barker. *Finite element solution of boundary value problems. Theory and computation.* Academic Press, Orlando, Fl., 1984. [Cited on page 258.]

[5] Owe Axelsson and Ben Polman. Block preconditioning and domain decomposition methods II. *J. Comp. Appl. Math.*, 24:55–72, 1988. [Cited on page 324.]

[6] David H. Bailey. Vector computer memory bank contention. *IEEE Trans. on Computers*, C-36:293–298, 1987. [Cited on page 36.]

[7] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986. [Cited on page 389.]

[8] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia PA, 1994. `http://www.netlib.org/templates/`. [Cited on page 260.]

[9] K.E. Batcher. MPP: A high speed image processor. In *Algorithmically Specialized Parallel Computers.* Academic Press, New York, 1985. [Cited on page 76.]

[10] Robert Beauwens and Mustafa Ben Bouzid. Existence and conditioning properties of sparse approximate block factorizations. *SIAM Numer. Anal.*, 25:941–956, 1988. [Cited on page 259.]

[11] Gordon Bell. The outlook for scalable parallel processing. Decision Resources, Inc, 1994. [Cited on page 72.]

[12] Abraham Berman and Robert J. Plemmons. *Nonnegative Matrices in the Mathematical Sciences.* SIAM, 1994. originally published by Academic Press, 1979, New York. [Cited on pages 202 and 445.]

[13] `https://en.wikipedia.org/wiki/Bfloat16_floating-point_format`. [Cited on page 188.]

[14] Petter E. Bjorstad, William Gropp, and Barry Smith. *Domain decomposition : parallel multilevel methods for elliptic partial differential equations.* Cambridge University Press, 1996. [Cited on page 318.]

[15] Fischer Black and Myron S Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–54, May-June 1973. [Cited on page 397.]

[16] Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical Report CMU-CS-93-173, CMU, 1993. [Cited on page 449.]

[17] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine cm-2. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, pages 3–16, New York, NY, USA, 1991. ACM. [Cited on page 370.]

[18] Mark Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *Solid-State Circuits Newsletter, IEEE*, 12(1):11 –13, winter 2007. [Cited on page 53.]

[19] Mark Bohr. The new era of scaling in an soc world. In *ISSCC*, pages 23–28, 2009. [Cited on page 55.]

[20] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, July 2003. [Cited on page 304.]

[21] BOOST interval arithmetic library. http://www.boost.org/libs/numeric/interval/doc/interval.htm. [Cited on page 187.]

[22] W. Briggs. *A Multigrid Tutorial*. SIAM, Philadelphia, 1977. [Cited on page 259.]

[23] S. Browne, J Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14:189–204, Fall 2000. [Cited on page 464.]

[24] A. Buluc and J. R. Gilbert. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, April 2008. [Cited on page 387.]

[25] A. W. Burks, H. H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Technical report, Harvard, 1946. [Cited on page 26.]

[26] A. Buttari, J. Dongarra, J. Langou, P. Luszczek, and J. Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *International Journal of High Performance Computing Applications*, 21:457–466, 2007. [Cited on pages 188 and 247.]

[27] Alfredo Buttari, Victor Eijkhout, Julien Langou, and Salvatore Filippone. Performance optimization and modeling of blocked sparse kernels. *Int. J. High Perf. Comput. Appl.*, 21:467–484, 2007. [Cited on pages 304 and 309.]

[28] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco. Dynamic octree load balancing using space-filling curves, 2003. [Cited on page 145.]

[29] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19:1749–1783, 2007. [Cited on pages 281 and 283.]

[30] A. P. Chandrakasan, R. Mehra, M. Potkonjak, J. Rabaey, and R. W. Brodersen. Optimizing power using transformations. *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, pages 13–32, January 1995. [Cited on page 55.]

[31] Chapel programming language homepage. http://chapel.cray.com/. [Cited on page 110.]

[32] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*, volume 10 of *Scientific Computation Series*. MIT Press, ISBN 0262533022, 9780262533027, 2008. [Cited on page 96.]

[33] A. Chronopoulos and C.W. Gear. *s*-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25:153–168, 1989. [Cited on pages 313 and 339.]

[34] Barry A. Cipra. An introduction to the ising model. *The American Mathematical Monthly*, pages 937–959, 1987. [Cited on page 398.]

[35] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32:406–242, 1953. [Cited on page 129.]

[36] Intel Corporation. bfloat16 - hardware numerics defintion. https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf, 2018. Document Number: 338302-001US. [Cited on pages 188 and 189.]

[37] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *ACM proceedings of the 24th National Conference*, 1969. [Cited on page 240.]

[38] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7(2):279–301, October 1989. [Cited on page 144.]

[39] Eduardo F. D'Azevedo, Mark R. Fahey, and Richard T. Mills. Vectorized sparse matrix multiply for compressed row storage format. *Lecture Notes in Computer Science, Computational Science ? ICCS 2005*, pages 99–106, 2005. [Cited on pages 232 and 304.]

[40] E.F. D'Azevedo, V.L. Eijkhout, and C.H. Romine. A matrix framework for conjugate gradient methods and some variants of cg with less synchronization overhead. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 644–646, Philadelphia, 1993. SIAM. [Cited on page 313.]

[41] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004. [Cited on pages 148 and 151.]

[42] J. Demmel, M. Heath, and H. Van der Vorst. Parallel numerical linear algebra. In *Acta Numerica 1993*. Cambridge University Press, Cambridge, 1993. [Cited on page 313.]

[43] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. Avoiding communication in sparse matrix computations. In *IEEE International Parallel and Distributed Processing Symposium*, 2008. [Cited on page 339.]

[44] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, R. Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93:293–312, February 2005. [Cited on page 309.]

[45] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256 – 268, oct 1974. [Cited on page 53.]

[46] Ashish Deshpande and Martin Schultz. Efficient parallel programming with linda. In *In Supercomputing '92 Proceedings*, pages 238–244, 1992. [Cited on page 112.]

[47] Tim Dettmers. 8-bit approximations for parallelism in deep learning. 11 2016. Proceedings of ICLR 2016. [Cited on page 188.]

[48] E. W. Dijkstra. Cooperating sequential processes. http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html. Technological University, Eindhoven, The Netherlands, September 1965. [Cited on page 90.]

[49] Edsger W. Dijkstra. Programming considered as a human activity. published as EWD:EWD117pub, n.d. [Cited on page 13.]

[50] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM Framework Supports Heterogeneous Network Computing. *Computers in Physics*, 7(2):166–75, April 1993. [Cited on page 105.]

[51] J. J. Dongarra. *The LINPACK benchmark: An explanation*, volume 297, chapter Supercomputing 1987, pages 456–474. Springer-Verlag, Berlin, 1988. [Cited on page 153.]

[52] Dr. Dobbs. Complex arithmetic: in the intersection of C and C++. http://www.ddj.com/cpp/184401628. [Cited on page 190.]

[53] Michael Driscoll, Evangelos Georganas, Penporn Koanantakool, Edgar Solomonik, and Katherine Yelick. A communication-optimal N-body algorithm for direct interactions. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 213. [Cited on pages 392 and 393.]

[54] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (release I). Technical Report RAL 92-086, Rutherford Appleton Laboratory, 1992. [Cited on page 230.]

[55] C. Edwards, P. Geng, A. Patra, and R. van de Geijn. Parallel matrix distributions: have we been doing it all wrong? Technical Report TR-95-40, Department of Computer Sciences, The University of Texas at Austin, 1995. [Cited on page 294.]

[56] Victor Eijkhout. A general formulation for incomplete blockwise factorizations. *Comm. Appl. Numer. Meth.*, 4:161–164, 1988. [Cited on page 324.]

[57] Victor Eijkhout. A theory of data movement in parallel computations. *Procedia Computer Science*, 9(0):236 – 245, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012, Also published as technical report TR-12-03 of the Texas Advanced Computing Center, The University of Texas at Austin. [Cited on page 393.]

[58] Victor Eijkhout, Paolo Bientinesi, and Robert van de Geijn. Towards mechanical derivation of Krylov solver libraries. *Procedia Computer Science*, 1(1):1805–1813, 2010. Proceedings of ICCS 2010, http://www.sciencedirect.com/science/publication?issn=18770509&volume=1&issue=1. [Cited on page 255.]

[59] Paul Erdös and A. Rényi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5:17?–61, 1960. [Cited on page 383.]

[60] V. Faber and T. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM J. Numer. Anal.*, 21:352–362, 1984. [Cited on page 258.]

[61] R.D. Falgout, J.E. Jones, and U.M. Yang. Pursuing scalability for hypre's conceptual interfaces. Technical Report UCRL-JRNL-205407, Lawrence Livermore National Lab, 2004. submitted to ACM Transactions on Mathematical Software. [Cited on page 311.]

[62] M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its applications to graph theory. *Czechoslovak Mathematical Journal*, 25:618–633, 1975. [Cited on page 445.]

[63] D. C. Fisher. Your favorite parallel algorithms might not be as fast as you think. *IEEE Trans. Computers*, 37:211–213, 1988. [Cited on page 64.]

[64] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21:948, 1972. [Cited on page 74.]

[65] Project fortress homepage. http://projectfortress.sun.com/Projects/Community. [Cited on page 111.]

[66] D. Frenkel and B. Smit. Understanding molecular simulations: From algorithms to applications, 2nd edition. 2002. [Cited on pages 347 and 350.]

[67] Roland W. Freund and Noël M. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numer. Math.*, 60:315–339, 1991. [Cited on page 259.]

[68] M. Frigo, Charles E. Leiserson, H. Prokop, and S. Ramachandran. Cache oblivious algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science*, 1999. [Cited on page 278.]

[69] Matteo Frigo and Volker Strumpen. The memory behavior of cache oblivious stencil computations. *J. Supercomput.*, 39(2):93–112, February 2007. [Cited on page 329.]

[70] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse gpu kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. [Cited on page 407.]

[71] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994. The book is available electronically, the url is `ftp://www.netlib.org/pvm3/book/pvm-book.ps`. [Cited on page 105.]

[72] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7:80–112, 1985. [Cited on page 112.]

[73] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992. [Cited on page 112.]

[74] Alan George. Nested dissection of a regular finite element mesh. 10(2):345–363, 1973. [Cited on page 318.]

[75] Andrew Gibiansky. Bringing HPC techniques to deep learning. https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/. [Cited on page 407.]

[76] GNU multiple precision library. http://gmplib.org/. [Cited on page 187.]

[77] D. Goldberg. Compuer arithmetic. Appendix in [100]. [Cited on page 171.]

[78] David Goldberg. What every computer scientist should know about floating-point arithmetic. *Computing Surveys*, March 1991. [Cited on pages 157 and 170.]

[79] G. H. Golub and D. P. O'Leary. Some history of the conjugate gradient and Lanczos algorithms: 1948-1976. 31:50–102, 1989. [Cited on page 253.]

[80] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, second edition edition, 1989. [Cited on page 212.]

[81] Google cloud TPU. https://cloud.google.com/tpu/. [Cited on page 302.]

[82] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):1–25, 2008. [Cited on page 300.]

[83] Ananth Y. Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel Distrib. Technol.*, 1(3):12–21, August 1993. [Cited on page 72.]

[84] F. Gray. Pulse code communication. U.S. Patent 2,632,058, March 17, 1953 (filed Nov. 1947). [Cited on page 125.]

[85] Ronald I. Greenberg and Charles E. Leiserson. Randomized routing on fat-trees. In *Advances in Computing Research*, pages 345–374. JAI Press, 1989. [Cited on page 128.]

[86] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1994. [Cited on page 100.]

[87] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. MIT Press, Nov. 2014. [Cited on page 100.]

[88] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference, Volume 2 - The MPI-2 Extensions*. MIT Press, 1998. [Cited on page 104.]

[89] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, USA, 1999. [Cited on page 100.]

[90] William Gropp, Thomas Sterling, and Ewing Lusk. *Beowulf Cluster Computing with Linux, 2nd Edition*. MIT Press, 2003. [Cited on page 78.]
[91] I. Gustafsson. A class of first-order factorization methods. *BIT*, 18:142–156, 1978. [Cited on page 259.]
[92] Diana Guttman, Meenakshi Arunachalam, Vlad Calina, and Mahmut Taylan Kandemir. Prefetch tuning optimizations. In James Reinders and Jim Jeffers, editors, *High Performance Pearls, Volume two*, pages 401–419. Morgan, 2015. [Cited on page 35.]
[93] Hadoop wiki. http://wiki.apache.org/hadoop/FrontPage. [Cited on page 148.]
[94] Louis A. Hageman and David M. Young. *Applied Iterative Methods*. Academic Press, New York, 1981. [Cited on page 246.]
[95] Mike Hamburg, Paul Kocher, and Mark E. Marson. Analysis of intel?s ivy bridge digital random number generator. Cryptography Research, Inc., http://www.rambus.com/wp-content/uploads/2015/08/Intel_TRNG_Report_20120312.pdf, 2012. [Cited on page 432.]
[96] A. Hartstein, V. Srinivasan, T. R. Puzak, and P. G. Emma. Cache miss behavior: is it $\sqrt{2}$? In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 313–320, New York, NY, USA, 2006. ACM. [Cited on page 457.]
[97] Michael T. Heath. *Scientific Computing: an introductory survey; second edition*. McGraw Hill, 2002. [Cited on pages 192, 212, and 216.]
[98] Don Heller. A survey of parallel algorithms in numerical linear algebra. *SIAM Review*, 20:740–777, 1978. [Cited on page 64.]
[99] B. A. Hendrickson and D. E. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Comput.*, 15(5):1201–1226, 1994. [Cited on page 289.]
[100] John L. Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufman Publishers, 3rd edition edition, 1990, 3rd edition 2003. [Cited on pages 12 and 480.]
[101] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, November 1999. [Cited on page 74.]
[102] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Nat. Bur. Stand. J. Res.*, 49:409–436, 1952. [Cited on page 253.]
[103] Catherine F. Higham and Desmond J. Higham. Deep learning: An introduction for applied mathematicians. *SIAM Review*, 61(4):860–891, 2019. [Cited on page 402.]
[104] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002. [Cited on pages 157, 175, 179, and 215.]
[105] Jonathan. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998. [Cited on page 114.]
[106] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. ISBN-10: 0131532715, ISBN-13: 978-0131532717. [Cited on page 74.]
[107] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. *SIGPLAN Not.*, 45(5):159–168, January 2010. [Cited on page 311.]
[108] Horovod homepage. https://horovod.ai/. [Cited on page 407.]
[109] Y. F. Hu and R. J. Blake. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, 25:417–444, 1999. [Cited on page 144.]
[110] IEEE 754: Standard for binary floating-point arithmetic. http://grouper.ieee.org/groups/754. [Cited on page 161.]

[111] IEEE 754-2019 standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. [Cited on pages 157, 161, and 167.]

[112] https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html. [Cited on page 302.]

[113] Interval arithmetic. http://en.wikipedia.org/wiki/Interval_(mathematics). [Cited on page 187.]

[114] C.R. Jesshope and R.W. Hockney editors. The DAP approach, volume 2. pages 311–329. Infotech Intl. Ltd., Maidenhead, 1979. [Cited on page 76.]

[115] M. T. Jones and P. E. Plassmann. The efficient parallel iterative solution of large sparse linear systems. In A. George, J.R. Gilbert, and J.W.H. Liu, editors, *Graph Theory and Sparse Matrix Computations*, IMA Vol 56. Springer Verlag, Berlin, 1994. [Cited on pages 327 and 440.]

[116] W. Kahan. Pracniques: Further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40–, January 1965. [Cited on page 187.]

[117] L. V. Kale and S. Krishnan. Charm++: Parallel programming with message-driven objects. In *Parallel Programming using C++, G. V. Wilson and P. Lu, editors*, pages 175–213. MIT Press, 1996. [Cited on page 113.]

[118] L.V. Kantorovich and G.P. Akilov. *Functional Analysis in Normed Spaces*. Pergamon press, 1964. [Cited on page 247.]

[119] R.M. Karp and Y. Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, 2-4 May 1988*, pages .290–300. ACM Press, 1988. [Cited on page 143.]

[120] J. Katzenelson. Computational structure of the n-body problem. *SIAM Journal of Scientific and Statistical Computing*, 10:787–815, July 1989. [Cited on page 391.]

[121] Kendall Square Research. http://en.wikipedia.org/wiki/Kendall_Square_Research. [Cited on page 113.]

[122] Donald Knuth. *The Art of Computer Programming, Volume 2: Seminumiercal algorithms*. Addison-Wesley, Reading MA, 3rd edition edition, 1998. [Cited on page 432.]

[123] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel spmd programs. In Keshav Pingali, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 331–345, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. [Cited on page 92.]

[124] Ulrich Kulisch. Very fast and exact accumulation of products. *Computing*, 91(4):397–405, April 2011. [Cited on page 179.]

[125] Ulrich Kulisch and Van Snyder. The exact dot product as basic tool for long interval arithmetic. *Computing*, 91(3):307–313, 2011. [Cited on page 179.]

[126] Milind Kulkarni, Martin Burtscher, Rajasekhar Inkulu, Keshav Pingali, and Calin Cascaval. How much parallelism is there in irregular applications? In *Principles and Practices of Parallel Programming (PPoPP)*, 2009. [Cited on page 83.]

[127] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing*. Benjamin Cummings, 1994. [Cited on page 363.]

[128] H.T. Kung. Systolic algorithms. In S. Parter, editor, *Large scale scientific computation*, pages 127–140. Academic Press, 1984. [Cited on page 19.]

[129] U. Kung, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. In *Proc. Intl. Conf. Data Mining*, pages 229–238, 2009. [Cited on page 381.]

[130] Christoph Lameter. NUMA (Non-Uniform Memory Access): An overview. 11, 2013. [Cited on page 94.]

[131] C. Lanczos. Solution of systems of linear equations by minimized iterations. *Journal of Research, Nat. Bu. Stand.*, 49:33–53, 1952. [Cited on page 253.]

[132] Rubin H Landau, Manual José Páez, and Cristian C. Bordeianu. *A Survey of Computational Physics.* Princeton University Press, 2008. [Cited on page 66.]

[133] J. Langou. *Iterative methods for solving linear systems with multiple right-hand sides.* Ph.D. dissertation, INSA Toulouse, June 2003. CERFACS TH/PA/03/24. [Cited on page 313.]

[134] Amy N. Langville and Carl D. Meyer. A survey of eigenvector methods for web information retrieval. *SIAM Review*, 47(1):135–161, 2005. [Cited on page 384.]

[135] P. L'Ecuyer. Combined multiple recursive generators. *Operations Research*, 44, 1996. [Cited on page 437.]

[136] R. B. Lehoucq. *Analysis and Implementation of an Implicitly Restarted Iteration.* PhD thesis, Rice University, Houston, TX, 1995. Also available as Technical report TR95-13, Dept. of Computational and Applied Mathematics. [Cited on page 260.]

[137] Charles E. Leiserson. Fat-Trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput*, C-34:892–901, 1985. [Cited on page 128.]

[138] Siyu Liao, Ashkan Samiee, Chunhua Deng, Yu Bai, and Bo Yuan. Compressing deep neural networks using toeplitz matrix: Algorithm design and fpga implementation. *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1443–1447, 2019. [Cited on page 405.]

[139] Stevel Lionel. Improving numerical reproducibility in C/C++/Fortran, 2013. [Cited on page 186.]

[140] Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM J. Numer. Anal.*, 16:346–358, 1979. [Cited on pages 321 and 323.]

[141] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36:177–189, 1979. [Cited on pages 318, 323, and 378.]

[142] J.D.C. Little. A proof of the queueing formula $L = \lambda W$. *Ope. Res.*, pages 383–387, 1961. [Cited on page 36.]

[143] Yongchao Liu, Douglas L. Maskell, and Bertil Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Res Notes*, 2:73, 2009. PMID- 19416548. [Cited on page 332.]

[144] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 4, 1986. [Cited on pages 327 and 440.]

[145] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 6–6, New York, NY, USA, 2009. ACM. [Cited on page 114.]

[146] M. Mascagni and A. Srinivasan. Algorithm 806: Sprng: A scalable library for pseudorandom number generation,. *ACM Transactions on Mathematical Software*, 26:436–461, 2000. [Cited on page 436.]

[147] M. Matsumoto and T. Nishimura. Dynamic creation of pseudorandom number generator. In E.H. Niederreiter and J. eds. Spanier, editors, *Monte Carlo and Quasi-Monte Carlo Methods*, pages 56–69. Springer, 2000. [Cited on page 437.]

[148] J.A. Meijerink and H.A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math Comp*, 31:148–162, 1977. [Cited on page 248.]

[149] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state

calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, June 1953. [Cited on page 398.]

[150] Gerard Meurant. Multitasking the conjugate gradient method on the CRAY X-MP/48. *Parallel Computing*, 5:267–280, 1987. [Cited on page 313.]

[151] Gérard Meurant. Domain decomposition methods for partial differential equations on parallel computers. *Int. J. Supercomputing Appls.*, 2:5–12, 1988. [Cited on page 324.]

[152] Kenneth Moreland and Ron Oldfield. Formal metrics for large-scale parallel performance. In Julian M. Kunkel and Thomas Ludwig, editors, *High Performance Computing*, volume 9137 of *Lecture Notes in Computer Science*, pages 488–496. Springer International Publishing, 2015. [Cited on page 73.]

[153] National Institute of Standards and Technology. Matrix market. http://math.nist.gov/MatrixMarket. [Cited on page 230.]

[154] Andrew T. Ogielski and William Aiello. Sparse matrix computations on parallel processor arrays. *SIAM J. Sci. Stat. Comput.* in press. [Cited on page 386.]

[155] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 30(5):2–11, September 1996. [Cited on page 39.]

[156] https://github.com/OpenMathLib/OpenBLAS. [Cited on page 302.]

[157] S. Otto, J. Dongarra, S. Hess-Lederman, M. Snir, and D. Walker. *Message Passing Interface: The Complete Reference*. The MIT Press, 1995. [Cited on page 104.]

[158] Michael L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM, Philadelphia PA, 2001. [Cited on page 157.]

[159] Larry Page, Sergey Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web, 1998. [Cited on page 384.]

[160] V. Ya. Pan. New combinations of methods for the acceleration of matrix multiplication. *Comp. & Maths. with Appls.*, 7:73–125, 1981. [Cited on pages 47 and 420.]

[161] Performance application programming interface. http://icl.cs.utk.edu/papi/. [Cited on page 464.]

[162] Seymour V. Parter. The use of linear graphs in Gaussian elimination. *SIAM Review*, 3:119–130, 1961. [Cited on page 234.]

[163] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J. Comput. Phys.*, 117:1–19, 1995. [Cited on page 350.]

[164] Christoph Pospiech. *High Performance Computing: 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings*, chapter Hunting Down Load Imbalance: A Moving Target, pages 497–505. Springer International Publishing, Cham, 2015. [Cited on page 143.]

[165] Siddhesh Poyarekar. Mostly harmless: An account of pseudo-normal floating point numbers. https://developers.redhat.com/blog/2021/05/12/mostly-harmless-an-account-of-pseudo-normal-floating-point-numbers/, 2021. [Cited on page 187.]

[166] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *Int'l Journal of High Performance Computing Applications*, 18(1):21–45, 2004. [Cited on page 276.]

[167] Gururaj S. Rao. Performance analysis of cache memories. *J. ACM*, 25:378–395, July 1978. [Cited on page 457.]

[168] J.K. Reid. On the method of conjugate gradients for the solution of large sparse systems of linear

equations. In J.K. Reid, editor, *Large sparse sets of linear equations*, pages 231–254. Academic Press, London, 1971. [Cited on page 253.]

[169] Y. Saad. *Iterative methods for sparse linear systems.* PWS Publishing Company, Boston, 1996. [Cited on page 260.]

[170] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: As easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery. [Cited on page 437.]

[171] Tetsuya Sato. The earth simulator: Roles and impacts. *Nuclear Physics B - Proceedings Supplements*, 129-130:102 – 108, 2004. Lattice 2003. [Cited on page 76.]

[172] David Schade. Canfar: Integrating cyberinfrastructure for astronomy. https://wiki.bc.net/atl-conf/display/BCNETPUBLIC/CANFAR+-+Integrating+Cyberinfrastructure+for+Astronomy. [Cited on page 149.]

[173] R. Schreiber. Scalability of sparse direct solvers. In A. George, J.R. Gilbert, and J.W.-H. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms (An IMA Workshop Volume).* Springer-Verlag, New York, 1993, 1993. also: Technical Report RIACS TR 92.13, NASA Ames Research Center, Moffet Field, Calif., May 1992. [Cited on page 289.]

[174] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018. [Cited on page 407.]

[175] D. E. Shaw. A fast, scalable method for the parallel evaluation of distance-limited pairwise particle interactions. *J. Comput. Chem.*, 26:1318–1328, 2005. [Cited on pages 351, 356, and 357.]

[176] D. B. Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and answers about BSP, 1996. [Cited on page 114.]

[177] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference, Volume 1, The MPI-1 Core.* MIT Press, second edition edition, 1998. [Cited on page 100.]

[178] Dan Spielman. Spectral graph theory, fall 2009. http://www.cs.yale.edu/homes/spielman/561/. [Cited on page 444.]

[179] Daniel A. Spielman and Shang-Hua Teng. Spectral partitioning works: Planar graphs and finite element meshes. *Linear Algebra and its Applications*, 421(2):284 – 305, 2007. [Cited on page 445.]

[180] Volker Springel. The cosmological simulation code GADGET-2. *Mon. Not. R. Astron. Soc.*, 364:1105–1134, 2005. [Cited on page 147.]

[181] G. W. Stewart. Communication and matrix computations on large message passing systems. *Parallel Computing*, 16:27–40, 1990. [Cited on page 289.]

[182] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969. [Cited on pages 47 and 420.]

[183] Robert Endre Tarjan. Amortized computational complexity. *SIAM J. Algebraic Discrete Methods*, 6(2):306?318, apr 1985. [Cited on page 421.]

[184] Universal Parallel C at George Washingon University. http://upc.gwu.edu/. [Cited on page 109.]

[185] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990. [Cited on pages 114 and 458.]

[186] Robert A. van de Geijn and Enrique S. Quintana-Ortí. *The Science of Programming Matrix Computations.* www.lulu.com, 2008. [Cited on pages 219 and 296.]

[187] Henk van der Vorst. A vectorizable variant of some ICCG methods. *SIAM J. Sci. Stat. Comput.*, 3:350–356, 1982. [Cited on page 336.]

[188] Henk van der Vorst. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13:631–644, 1992. [Cited on page 259.]

[189] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33, June 2015. [Cited on page 302.]

[190] Richard S. Varga. *Matrix Iterative Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1962. [Cited on pages 244 and 253.]

[191] R. Vuduc, J. Demmel, and K. Yelikk. Oski: A library of automatically tuned sparse matrix kernels. In *(Proceedings of SciDAC 2005, Journal of Physics: Conference Series, to appear.*, 2005. [Cited on page 309.]

[192] Richard W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California Berkeley, 2003. [Cited on page 304.]

[193] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 12–21, New York, NY, USA, 1993. ACM. [Cited on page 147.]

[194] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`). [Cited on page 276.]

[195] O. Widlund. On the use of fast methods for separable finite difference equations for the solution of general elliptic problems. In D.J. Rose and R.A. Willoughby, editors, *Sparse matrices and their applications*, pages 121–134. Plenum Press, New York, 1972. [Cited on page 248.]

[196] J.H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, N.J., 1963. [Cited on pages 157, 175, and 215.]

[197] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52:65–76, April 2009. [Cited on page 47.]

[198] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995. [Cited on pages 21 and 463.]

[199] L. T. Yand and R. Brent. The improved bicgstab method for large and sparse unsymmetric linear systems on parallel distributed memory architectures. In *Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing*. IEEE, 2002. [Cited on page 313.]

[200] Randy Yates. Fixed point: an introduction. `http://www.digitalsignallabs.com/fp.pdf`, 2007. [Cited on page 189.]

[201] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 25–, Washington, DC, USA, 2005. IEEE Computer Society. [Cited on page 387.]

[202] David M. Young. *Iterative method for solving partial differential equations of elliptic type*. PhD thesis, Harvard Univ., Cambridge, MA, 1950. [Cited on page 253.]

[203] L. Zhou and H. F. Walker. Residual smoothing techniques for iterative methods. *SIAM J. Sci. Stat. Comput.*, 15:297–312, 1994. [Cited on page 431.]

ISBN 978-1-257-99254-6

