Introduction to High Performance Scientific Computing

The Art of HPC, volume 1

Evolving Copy - open for comments

Victor Eijkhout
with
Edmond Chow, Robert van de Geijn

3rd edition 2022, formatted December 1, 2022
Series reference: https://theartofhpc.com

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

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 1 *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 6.12.*

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 2 *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 3 *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. 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 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.

1.2.1 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.

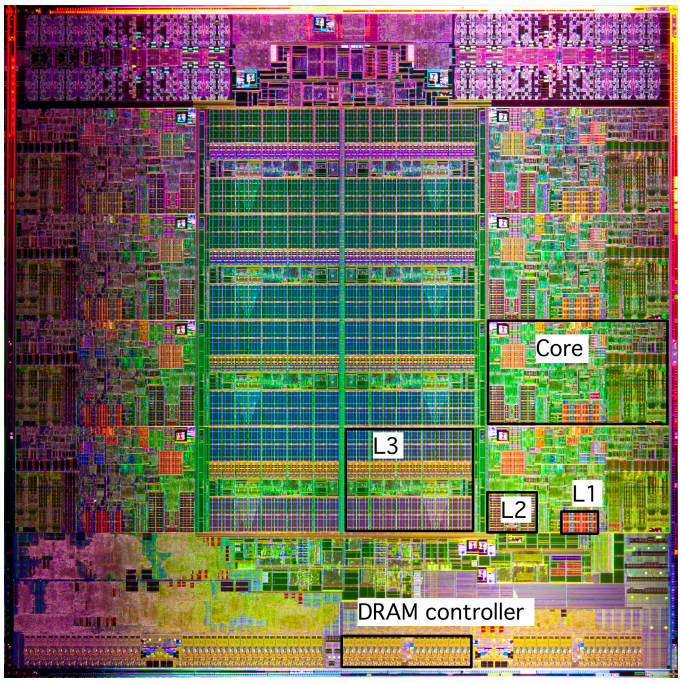


Figure 1.1: The Intel Sandy Bridge processor die.

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.

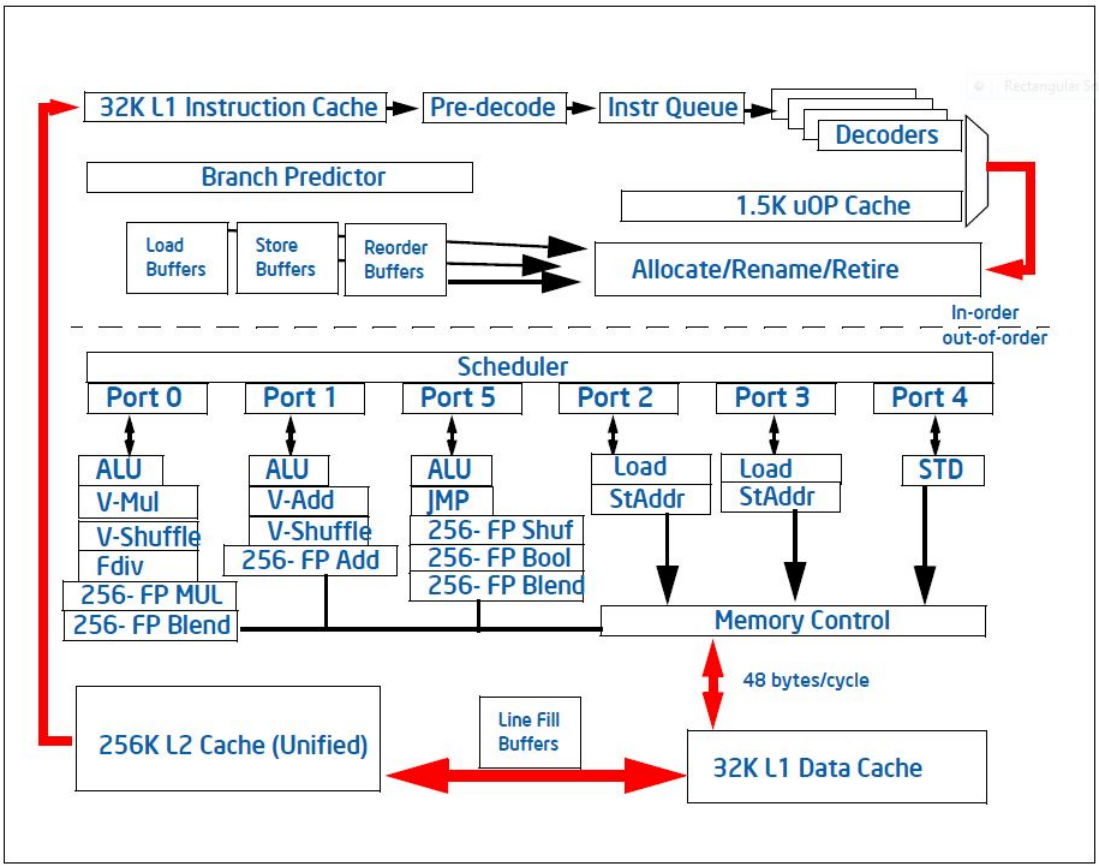


Figure 2-1. Intel microarchitecture code name Sandy Bridge Pipeline Functionality

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.

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.

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 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.

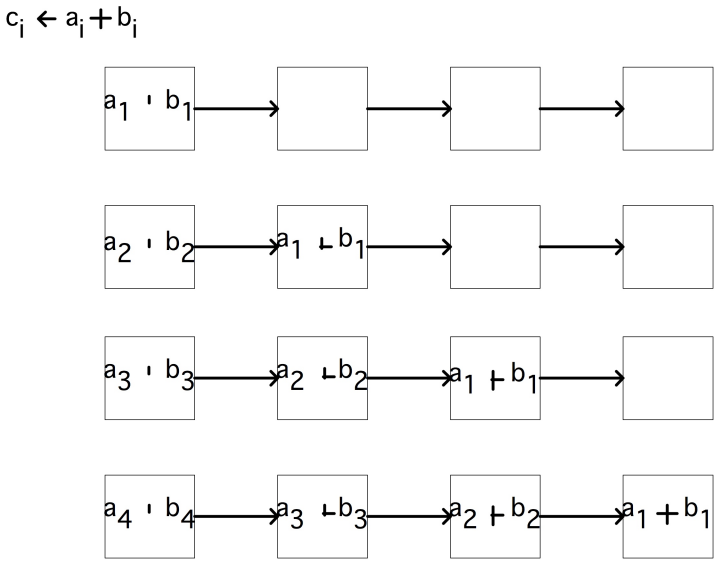


Figure 1.3: Schematic depiction of a pipelined operation.

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 (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 [126]. This is also connected to computing with Field-Programmable Gate Arrays (FPGAs), where the systolic array is software-defined.

Section 1.7.3 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 1.3.4.

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 1.3.5).

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*. Recall from section 1.1 that our 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 is unrealistic because of the so-called *memory wall* [192], also known as the *von Neumann bottleneck*: memory is too slow to load data into the process at the rate the processor can absorb it. Specifically, a single load can take 1000 cycles, while a processor can perform several operations per cycle. (After this long wait for a load, the next load can come faster, but still too slow for the processor. This matter of wait time versus throughput will be addressed below in section 1.3.2.)

To solve this bottleneck problem, a modern processor 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; in this section we will mostly go into the technical aspects.

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 in this chapter, and particularly section 1.7.

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.

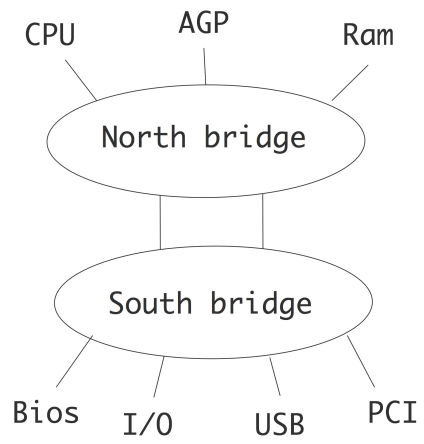


Figure 1.4: Bus structure of a processor.

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.

Typically, the further away from the processor one gets, the longer the latency is, and the lower the bandwidth. These two factors make 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. To illustrate that this is a serious matter, consider a vector addition

```
for (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, `a[i]` is loaded before it can be written, so there are 4 memory access, 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 operations, and in fact will have no effect on the above example. Strategies for programming that lead to significant cache use are discussed in section 1.7.

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

1.3.3 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
```

is actually 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 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), you see the explicit load, compute, and store instructions.

Compute instructions such as add or multiply only operate on registers. For instance, in *assembly language* you will see 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. For instance, if the above computation is followed by a statement

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

the computed value of a could be left in register. This optimization is typically performed as a *compiler optimization*: the compiler will simply not generate the instructions for storing and reloading a. We say that a stays *resident in register*.

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
```

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.

1.3.4 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 sectionsec:coding-cachesize.

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.

1.3.4.1 A motivating example

As an example, let's suppose a variable x is used twice, and its uses are too far apart that it would stay *resident in 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.

Since loading from cache is faster than loading from main memory, the computation will now be faster. Caches are fairly small, so values can not be kept there indefinitely. We will see the implications of this in the following discussion.

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. Thus cache use and reuse is outside of direct programmer control. Later, especially in sections 1.6.2 and 1.7, you will see how it is possible to influence cache use indirectly.

1.3.4.2 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.

1.3.4.3 Cache levels, speed, and size

The caches are called 'level 1' and 'level 2' (or, for short, L1 and L2) cache; some processors can have an L3 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; the L3 cache is off-chip. The L1 cache is small, typically around 16 Kbyte. Level 2 (and, when present, level 3) cache is more plentiful, up to several megabytes, but it is also slower. Unlike main memory, which is expandable, caches are fixed in size. If a version of a processor chip exists with a larger cache, it is usually considerably more expensive.

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 main memory. Finding data in cache is called a *cache hit*, and not finding it a *cache miss*.

Figure 1.5 illustrates the basic facts of the *cache hierarchy*, in this case for the *Intel Sandy Bridge* chip: the closer caches are to the FPUs, the faster, but also the smaller they are. Some points about this figure.

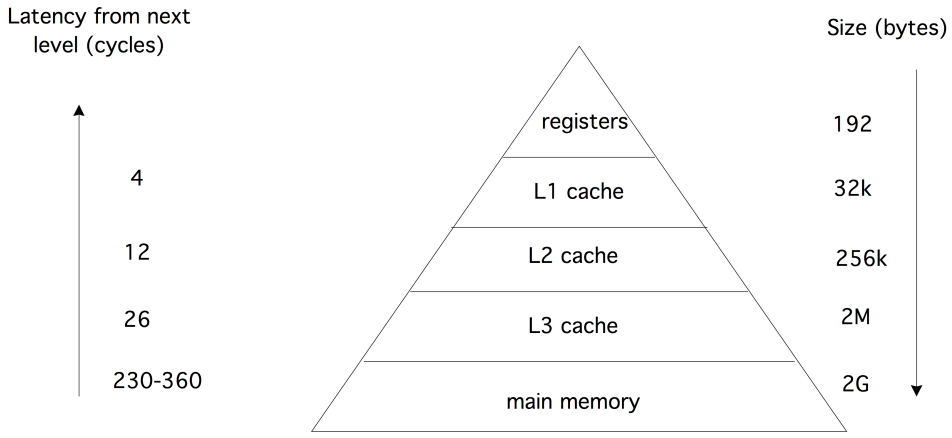


Figure 1.5: Memory hierarchy of an Intel Sandy Bridge, characterized by speed and size.

- Loading data from registers is so fast that it does not constitute a limitation on algorithm execution speed. On the other hand, there are few registers. Each core has 16 general purpose registers, and 16 SIMD registers.
- The L1 cache is small, but sustains a bandwidth of 32 bytes, that is 4 double precision number, per cycle. This is enough to load two operands each for two operations, but note that the core can actually perform 4 operations per cycle. Thus, to achieve peak speed, certain operands need to stay in register: typically, L1 bandwidth is enough for about half of peak performance.

- The bandwidth of the L2 and L3 cache is nominally the same as of L1. However, this bandwidth is partly wasted on coherence issues.
- Main memory access has a latency of more than 100 cycles, and a bandwidth of 4.5 bytes per cycle, which is about 1/7th of the L1 bandwidth. However, this bandwidth is shared by the multiple cores of a processor chip, so effectively the bandwidth is a fraction of this number. Most clusters will also have more than one *socket* (processor chip) per node, typically 2 or 4, so some bandwidth is spent on maintaining *cache coherence* (see section 1.4), again reducing the bandwidth available for each chip.

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 1.7.4 and 24.2.

1.3.4.4 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 1.3.5 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. (Section 1.3.4.6 discusses how the processor 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 1.3.4.10.

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.

1.3.4.5 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 1.7.13 and exercise 1.26.

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.

1.3.4.6 Replacement policies

Data in cache and registers is placed there by the system, outside of programmer control. Likewise, the system decides when to overwrite data in the cache or in registers 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 from cache*, meaning that it is overwritten with the new item, and 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.

1.3.4.7 Cache lines

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 *cache line*, sometimes called a *cache block*. A typical cache line is 64 or 128 bytes long, which in the context of scientific computing implies 8 or 16 double precision floating point numbers. The cache line size for data moved into L2 cache can be larger than for data moved into L1 cache.

A first motivation for cache lines is a practical one of simplification: if a cacheline is 64 bytes long, six fewers 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. See section 1.6.2 for discussion.

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 (see section 1.7.5 for some examples). An efficient program then tries to use the other items on the cache line, 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.

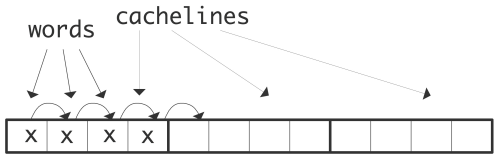


Figure 1.6: Accessing 4 elements at stride 1.

Stride 1 corresponds to sequential access of an array:

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

Let us use as illustration a 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

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

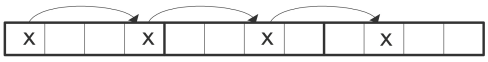


Figure 1.7: Accessing 4 elements at stride 3.

implies that in every cache line only certain elements are used. We illustrate that with stride 3: requesting the first elements 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 needed loading three cache lines 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 a hardware mechanism that notices the regular access patterns, and pre-emptively loads further cachelines; see section 1.3.5.)

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 3.8.7). Also, methods that use *recursive doubling* often have a code structure that exhibits non-unit strides

```
for (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 24.1.3 discusses ways to address *cacheline boundary alignment* in practice.

1.3.4.8 *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 1.3.4.6 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'.

1.3.4.9 *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 1.8.

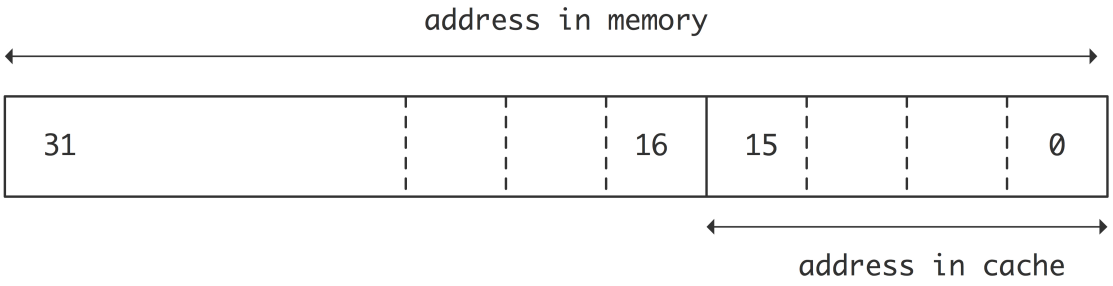


Figure 1.8: Direct mapping of 32-bit addresses into a 64K cache.

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 (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.

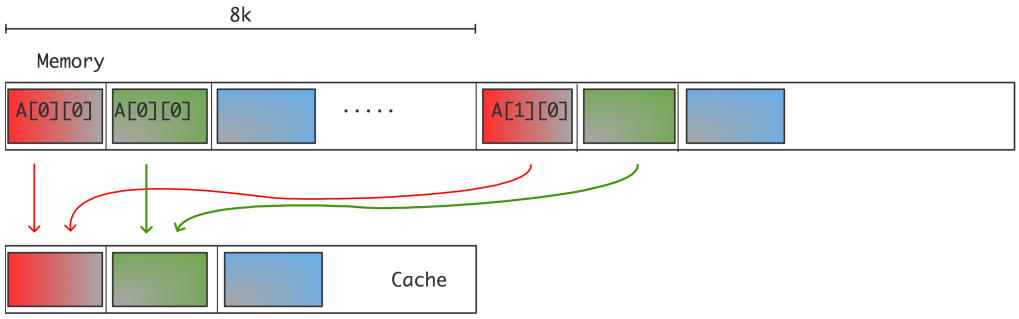


Figure 1.9: Mapping conflicts in direct mapped cache.

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 1.9.

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 cache line 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 cache line, 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 cache line containing `a[2][0]` is brought into memory. This is again mapped to the same location, causing flushing of the cache line just loaded for `a[1][0]`.
- In the next iteration, `a[0][1]` is needed, which is on the same cache line as `a[0][0]`. However, this cache line 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 cache line 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 cache line 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 4 *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.

1.3.4.10 Associative caches

The problem of cache conflicts, outlined in the previous section, 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 1.3.4.6) would flush data to make room for the incoming item. Such a cache is called *fully associative*, and while it seems optimal, it is also very costly to build, and much 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.

In this section we explore the idea of associativity; practical aspects of cache associativity are explored in section 1.7.7.

Code would have to have a $k+1$-way conflict before data would be flushed prematurely as in the above example. In that example, a value of $k = 2$ would suffice, but in practice higher values are often encountered. Figure 1.10 illustrates the mapping of memory addresses to cache locations for a direct mapped

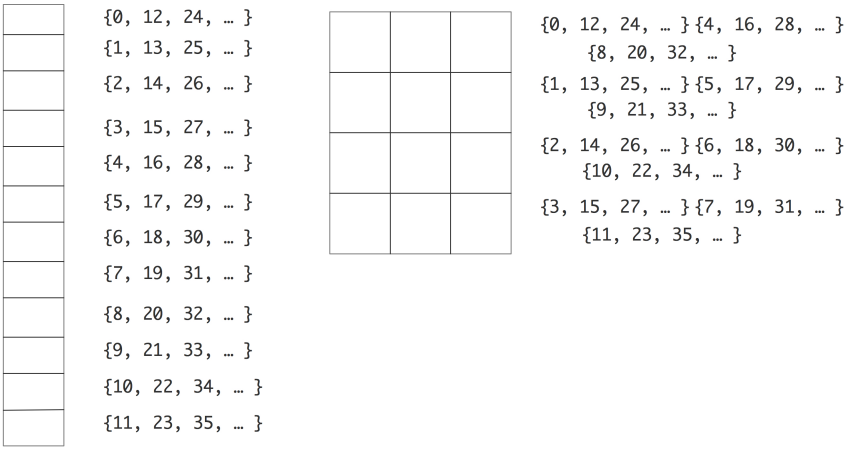


Figure 1.10: Two caches of 12 elements: direct mapped (left) and 3-way associative (right).

and a 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 Woodcrest* processor has an L1 cache of 32K bytes that is 8-way set associative with a 64 byte cache line size, and an L2 cache of 4M bytes that is 8-way set associative with a 64 byte cache line size. On the other hand, the *AMD Barcelona* chip has 2-way associativity for the L1 cache, and 8-way for the L2. A higher associativity ('way-ness') is obviously desirable, but makes a processor slower, since determining whether an address is already in cache becomes more complicated. For this reason, the associativity of the L1 cache, where speed is of the greatest importance, is typically lower than of the L2.

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

1.3.4.11 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.

1.3.4.12 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.

1.3.5 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, often subsequent data items are 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 1.11.

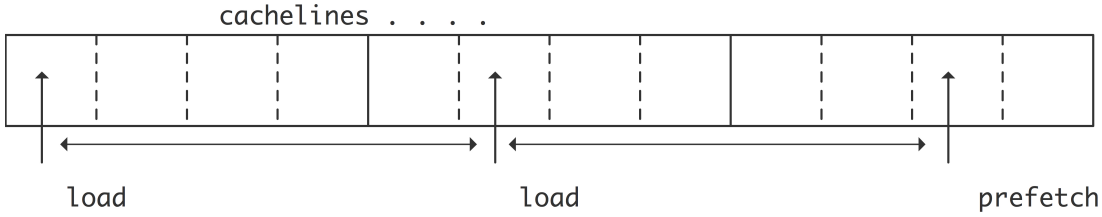


Figure 1.11: Prefetch stream generated by equally spaced requests.

In its simplest form, the CPU will detect that consecutive loads come from two consecutive cache lines, and automatically issue a request for the next following cache line. This process can be repeated or extended if the code makes an actual request for that third cache line. Since these cache lines 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.

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.

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.

1.3.6 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* [140]:

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

This is illustrated in figure 1.12. The problem with maintaining this concurrency is not that a program does 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 1.3.5) 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

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

1.3.7 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

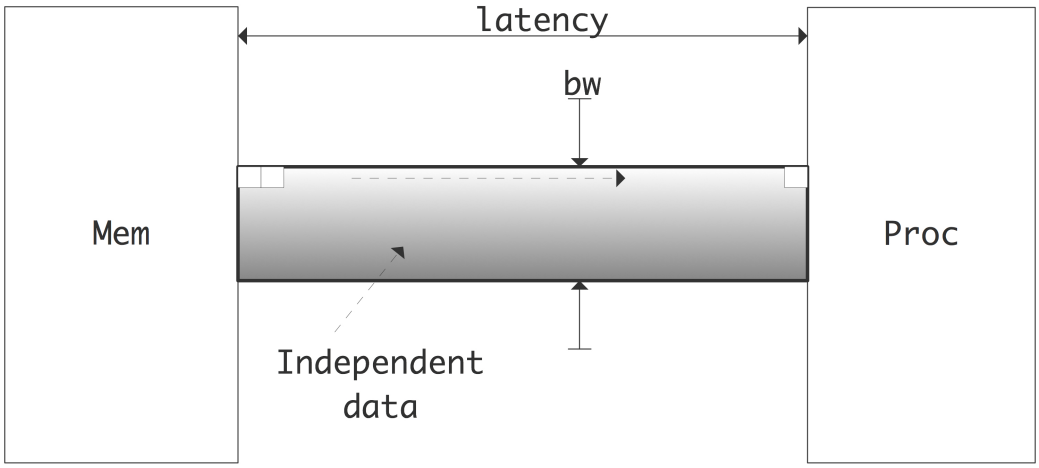


Figure 1.12: Illustration of Little's Law that states how much independent data needs to be in flight.

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* [7]. 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 (s=2; s<2*n; s*=2)
  for (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 (s=(n+1)/2; s>1; s/=2)
  for (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 cache lines 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 cache line is efficient, but strided access suffers from a deteriorated performance.

1.3.8 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, and this translation has to be dynamic. A program has a 'logical data space' (typically starting from address zero) of the addresses used in the compiled code, and this needs to be 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.

1.3.8.1 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.

1.3.8.2 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 consulted, if necessary bringing the needed page into memory. The TLB is (sometimes fully) associative (section 1.3.4.10), using an LRU policy (section 1.3.4.6).

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 1.7.6 and appendix 24.5 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 1.8.1.
- 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 introducing what is in effect data parallelism (section 2.3.1), collaborate on a common task at a higher overall efficiency[153].

Remark 5 *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 simply two processors on the same die, later generations incorporated L3 or L2 caches that were shared between the two processor cores; see figure 1.13. This design makes it

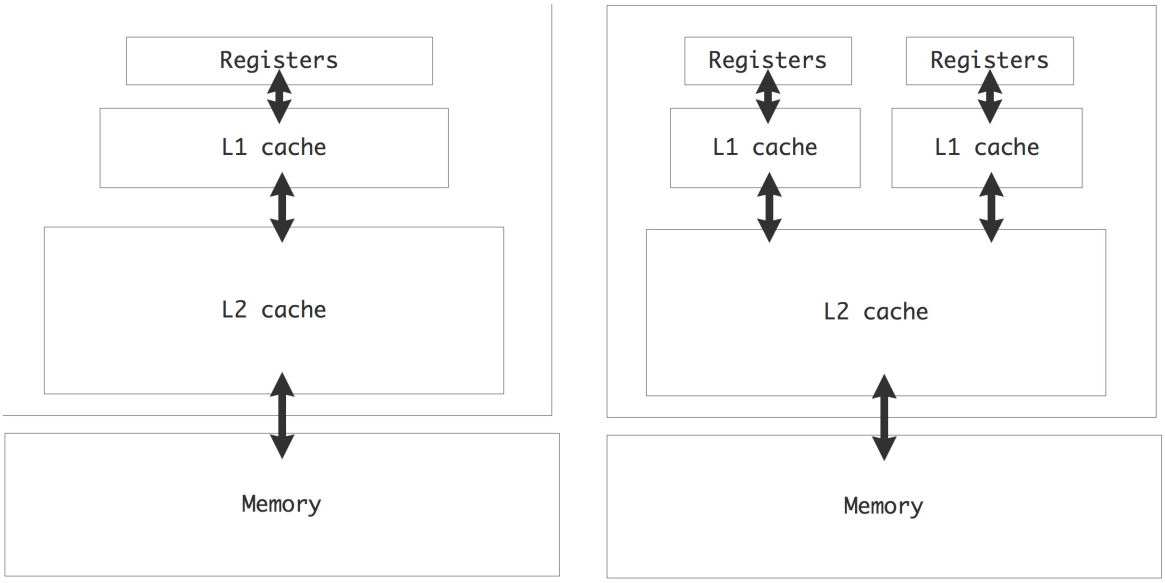


Figure 1.13: Cache hierarchy in a single-core and dual-core chip.

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 4 or 6 cores are common, even in laptops, and Intel and AMD are marketing 12-core chips. The core count is likely to go up in the future: Intel has already shown an 80-core prototype that is developed into the 48 core 'Single-chip Cloud Computer', illustrated in fig 1.14. This chip has a structure with 24 dual-core 'tiles' that are connected through a 2D mesh network. Only certain tiles are connected to a memory controller, others can not reach memory other than through the on-chip network.

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

Core The cores have their own private L1 cache, which is a sort of distributed memory. The above mentioned Intel 80-core prototype has the cores communicating in a distributed memory fashion.

Socket On one socket, there is often a shared L2 cache, which is shared memory for 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.

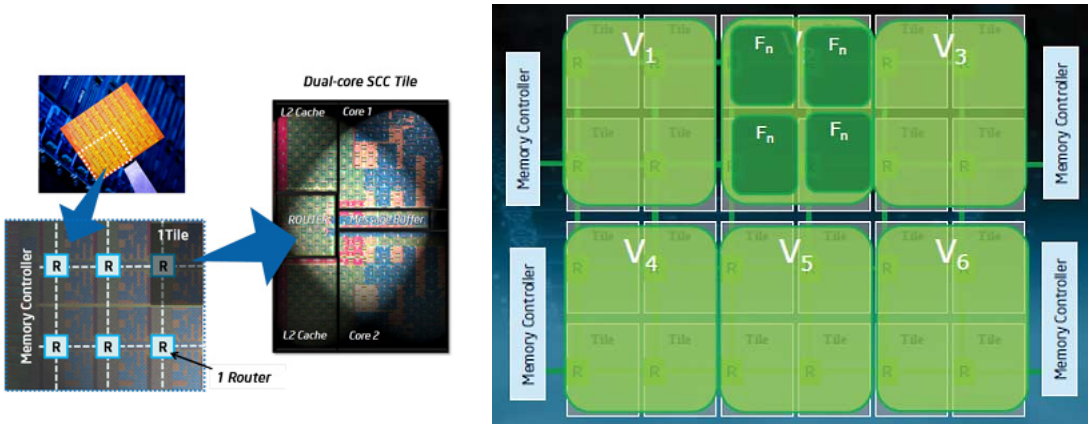


Figure 1.14: Structure of the Intel Single-chip Cloud Computer chip.

1.4.1 Cache coherence

With parallel processing, there is the potential for a conflict if more than one processor 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 one processor alters its copy, the other copy needs to be updated.

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, processors typically have some private cache that contains copies of data from memory, so conflicting copies can occur. This situation arises in particular in multicore designs.

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 needed. (This makes updating memory locations an *atomic operation*; more about this in section 2.6.1.5.) However, it will slow down the computation, and it wastes bandwidth to the core that could otherwise be used for loading or storing operands.

The state of a cache line with respect to a data item in main memory is usually described as one of the following:

Scratch: the cache line does not contain a copy of the item;
Valid: the cache line is a correct copy of data in main memory;
Reserved: the cache line is the *only* copy of that piece of data;

Dirty: the cache line has been modified, but not yet written back to main memory;

Invalid: the data on the cache line is also present on other processors (it is not *reserved*), and another process has modified its copy of the data.

A simpler variant of this is the Modified-Shared-Invalid (MSI) coherence protocol, where a cache line 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. (This list of transitions is a *Finite State Automaton (FSA)*; see section 20.)

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) ...
}
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:

```
// 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 6.12.

1.4.4 TLB shootdown

Section 1.3.8.2 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 node is typically spread over banks that are directly attached to one particular socket. This is for instance illustrated in figure 1.15, 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.

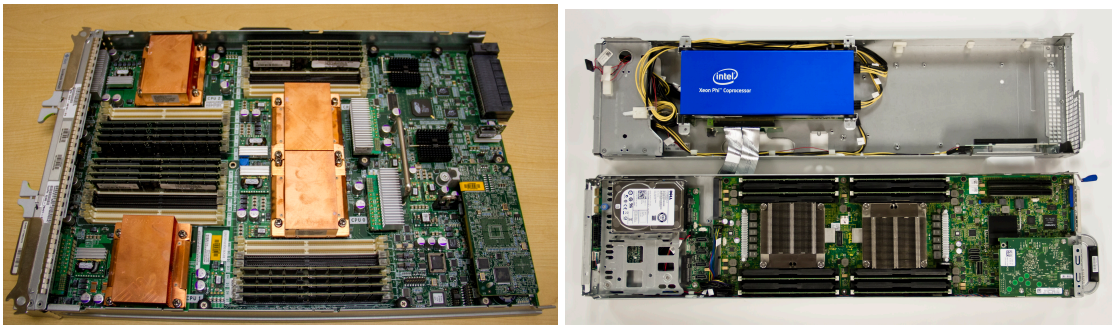


Figure 1.15: Left: a four-socket design. Right: a two-socket design with co-processor.

- 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 7) 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 somewhat unbalanced: 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 $f(n)/n$.

(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 a reuse 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 reuse here is 1.

1.6.1.2 Example: matrix operations

Next, consider the *matrix-matrix product*:

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

This 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 operation*, clearly has data reuse by the above definition. Argue that this reuse is not trivially attained by a simple implementation. What determines whether the naive 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 [178, 157]. However, this requires a different implementation, which has its own analysis in terms of memory access and reuse.]

The matrix-matrix product is the heart of the *LINPACK benchmark* [51]; see section 2.11.4. Using this as the sole measure of *benchmarking* a computer may give an optimistic view of its performance: the matrix-matrix product is an operation that has considerable data reuse, so it is relatively insensitive to memory bandwidth and, for parallel computers, properties of the network. Typically, computers will attain 60–90% of their *peak performance* on the Linpack benchmark. Other benchmark may give considerably lower figures.

1.6.1.3 The roofline model

There is an elegant way of talking about how arithmetic intensity, which is a statement about the ideal algorithm, not its implementation, interacts with hardware parameters and the actual implementation to determine performance. This is known as the *roofline model* [191], and it expresses the basic fact that performance is bounded by two factors, illustrated in the first graph of figure 1.16.

1. The *peak performance*, indicated by the horizontal line at the top of the graph, is an absolute bound on the performance[3], 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.

3. An old joke states that the peak performance is that number that the manufacturer guarantees you will never exceed.

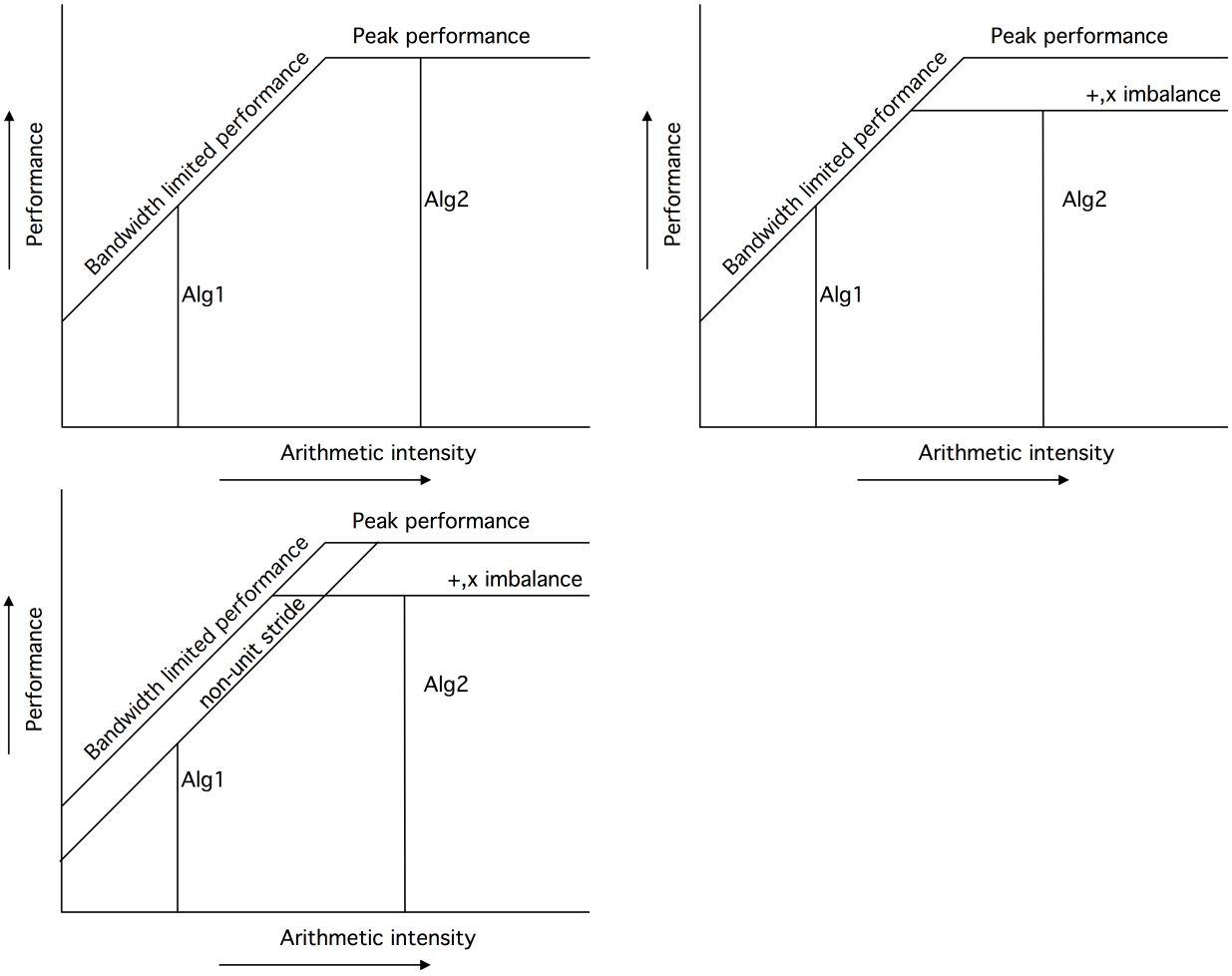


Figure 1.16: Illustration of factors determining performance in the roofline model.

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

$$\frac{operations}{second} = \frac{operations}{data\ item} \cdot \frac{data\ items}{second}$$

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 1.16 indicates various factors that lower the ceiling. There are also various factors that lower the available bandwidth, such as imperfect data 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.

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 cheaper than getting data from main memory, a programmer obviously wants to code in such a way that data in cache is reused. While placing data in cache is not under explicit programmer control, even from assembly language (low level memory access can be controlled by the programmer in the Cell processor and in some GPUs.), in most CPUs, it is still possible, knowing the behavior of the caches, to know what data is in cache, and to some extent to control it.

The two crucial concepts here 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. Since most caches have an LRU replacement policy (section 1.3.4.6), if in between the two references less data has been referenced than the cache size, the element will still be in cache and therefore be quickly accessible. With other replacement policies, such as random replacement, this guarantee can not be made.

1.6.2.1 Temporal locality

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

```
for (loop=0; loop<10; loop++) {
  for (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. Therefore, the use of x[i] does not exhibit temporal locality: subsequent uses are spaced too far apart in time for it to remain in cache.

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

```
for (i=0; i<N; i++) {
  for (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].

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. Above, you have seen two examples of spatial locality:

- 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 (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 1.7.5.
- A second example of spatial locality worth observing involves the TLB (section 1.3.8.2). 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 1.7.6.

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 (k=0; k<nvectors; k++)
  for (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 (i=0; i<length; i++)
  for (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 1.17 The first implementation constructs the (i, j) element

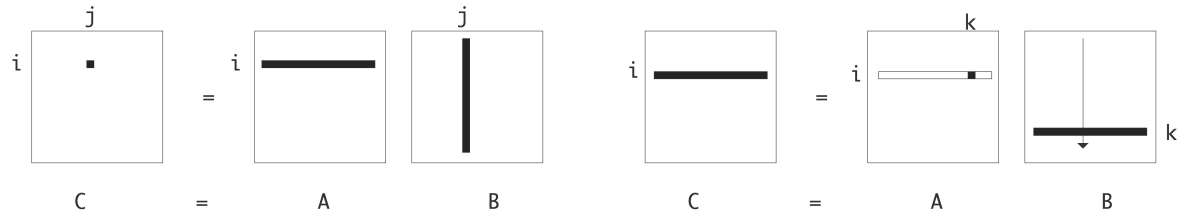


Figure 1.17: Two loop orderings for the $C \leftarrow A \cdot B$ matrix-matrix product.

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.

`c[i,j]` 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.

`a[i,k]` 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.

`b[k,j]` 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.

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*.

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. The issue here is that of *cache coherence* (section 1.4.1) where 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 is not just a property of a program, but also to a large extent of how the program is executed in parallel.

1.7 Programming strategies for high 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 24.

1.7.1 Peak performance

For marketing purposes, it may be desirable to define a 'top speed' for a CPU. 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), number of floating point units, and the number of cores; see section 1.4. This top speed is unobtainable in practice, and very few codes come even close to it. The *Linpack benchmark* is one of the measures how close you can get to it; the parallel version of this benchmark is reported in the 'top 500'; see section 2.11.4.

1.7.2 Bandwidth

You have seen in section 1.6.1.3 that algorithms can be *bandwidth-bound*, meaning that they are more limited by the available bandwidth than by available processing power. In that case we can wonder if the amount of bandwidth per core is a function of the number of cores: it is conceivable that the total bandwidth is less than the product of the number of cores and the bandwidth to a single core.

We test this on TACC's *Frontera* cluster, with dual-socket *Intel Cascade Lake* processors, with a total of 56 cores per node.

We let each core execute a simple streaming kernel:

```
// allocation.cxx
template <typename R>
R Cache<R>::sumstream(int repeats,size_t length,size_t byte_offset /* =0 default */ ) const {
  R s{0};
  size_t loc_offset = byte_offset/sizeof(R);
  const R *start_point = thecache.data()+loc_offset;
  for (int r=0; r<repeats; r++)
    for (int w=0; w<length; w++)
      s += *( start_point+w ) * r;
  return s;
};
```

and execute this on different threads and disjoint memory locations:

```
// bandwidth.cxx
    vector<double> results(nthreads,0.);
    for ( int t=0; t<nthreads; t++) {
      auto start_point = t*stream_length;
      threads.push_back
        ( thread( [=,&results] () {
            results[t] = memory.sumstream(how_many_repeats,stream_length,start_point);
          } ) );
    }
    for ( auto &t : threads )
      t.join();
```

We see that for 40 threads there is essentially no efficiency loss, but with 56 threads there is a 10 percent loss.

```
OMP_PROC_BIND=true ./bandwidth -t 56 -s 8 -k 64
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                Processing 8192 words
                over up to 56 threads
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Overhead for 1 threads:       66 usec
Threads  1.. ran for   136965 usec (efficiency: 100% )
Overhead for 8 threads:      187 usec
Threads  8.. ran for   137492 usec (efficiency: 99% )
Overhead for 16 threads:      260 usec
Threads 16.. ran for   137642 usec (efficiency: 99% )
Overhead for 24 threads:      406 usec
Threads 24.. ran for   137930 usec (efficiency: 99% )
Overhead for 32 threads:      559 usec
Threads 32.. ran for   138183 usec (efficiency: 99% )
Overhead for 40 threads:      793 usec
Threads 40.. ran for   138405 usec (efficiency: 98% )
Overhead for 48 threads:     1090 usec
Threads 48.. ran for   143611 usec (efficiency: 95% )
Overhead for 56 threads:     1191 usec
Threads 56.. ran for   149732 usec (efficiency: 91% )
```

1.7.3 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.

1.7.3.1 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
```

1.7.4 Cache size

Above, you learned that data from L1 can be moved with lower latency and higher bandwidth than from L2, and L2 is again faster than L3 or memory. This is easy to demonstrate with code that repeatedly accesses the same data:

```
for (i=0; i<NRUNS; i++)
  for (j=0; j<size; j++)
    array[j] = 2.3*array[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 1.19. Argue that with a large enough problem and an LRU replacement policy (section 1.3.4.6) 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?

Often, it is possible to arrange the operations to keep data in L1 cache. For instance, in our example, we could write

```
for (b=0; b<size/l1size; b++) {
  blockstart = 0;
  for (i=0; i<NRUNS; i++) {
    for (j=0; j<l1size; j++)
      array[blockstart+j] = 2.3*array[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 6 *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 allowed to make.*

1.7.4.1 Measuring cache performance at user level

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 unforesoon 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++)
    memory[iword] += 1.1;
}
```

may seem to be independent of the array size.

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 prevent this loop exchange, you can try to make the inner loop more too complicated for the compiler to analyze. 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++)
    memory[iword] = (iword+1) % cachesize_in_words
```

```
// use:
ptr = 0
for (int iword=0; iword<cachesize_in_words; iword++)
    ptr = memory[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* 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++)
    memory[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.

The code for this is given in section 24.2.

Exercise 1.20. 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?

1.7.4.2 *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 1.18. The full code is given in section 24.2.

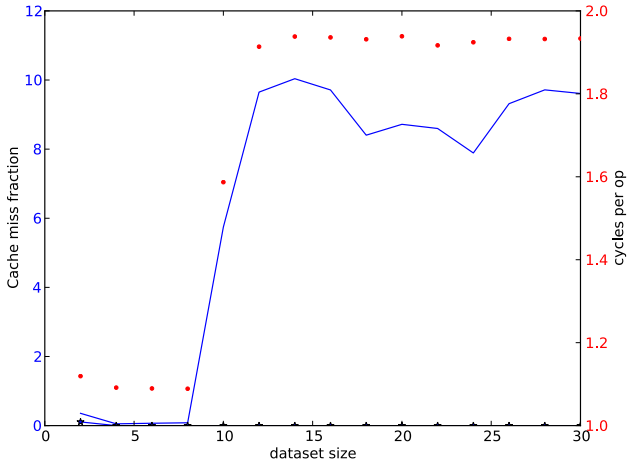


Figure 1.18: Average cycle count per operation as function of the dataset size.

1.7.5 Cache lines and striding

Since data is moved from memory to cache in consecutive chunks named cachelines (see section 1.3.4.7), code that does not utilize all data in a cacheline pays a bandwidth penalty. This is born out by a simple code

```
for (i=0,n=0; i<L1WORDS; i++,n+=stride)
  array[n] = 2.3*array[n]+1.2;
```

Here, a fixed number of operations is performed, but on elements that are at distance `stride`. As this *stride* increases, we expect an increasing runtime, which is born out by the graph in figure 1.19.

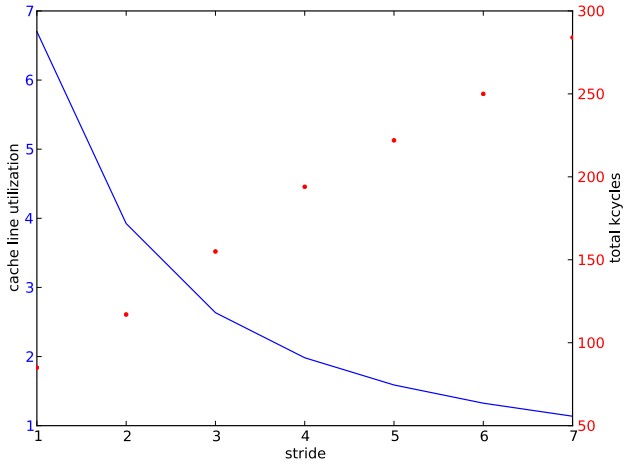


Figure 1.19: Run time in kcycles and L1 reuse as a function of stride.

The graph also shows a decreasing reuse of cachelines, defined as the number of vector elements divided by the number of L1 misses (on stall; see section 1.3.5).

The full code is given in section 24.3.

The effects of striding can be mitigated by the bandwidth and cache behavior of a processor. Consider some run on the *Intel Cascade Lake* processor of the *Frontera* cluster at TACC, (28-cores per socket, dual socket, for a total of 56 cores per node). We measure the time-per-operation on a simple streaming kernel, using increasing strides. Table 1.2 reports in the second column indeed a per-operation time that goes up linearly with the stride.

However, this is for a dataset that overflows the L2 cache. If we make this run contained in the L2 cache, as reported in the 3rd column, this increase goes away as there is enough bandwdidth available to stream strided data at full speed from L2 cache.

stride	nsec/word 56 cores, 3M	56 cores, 0.3M	28 cores, 3M
1	7.268	1.368	1.841
2	13.716	1.313	2.051
3	20.597	1.319	2.852
4	27.524	1.316	3.259
5	34.004	1.329	3.895
6	40.582	1.333	4.479
7	47.366	1.331	5.233
8	53.863	1.346	5.773

Table 1.2: Time per operation in nanoseconds as a function of striding on 56 cores of Frontera, per-core datasize 3.2M.

1.7.6 TLB

As explained in section 1.3.8.2, 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 24.5 for the source code) are plotted in figures 1.22 and 1.21.

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 1.21 shows that, with increasing n, the number of TLB misses per column is roughly two.

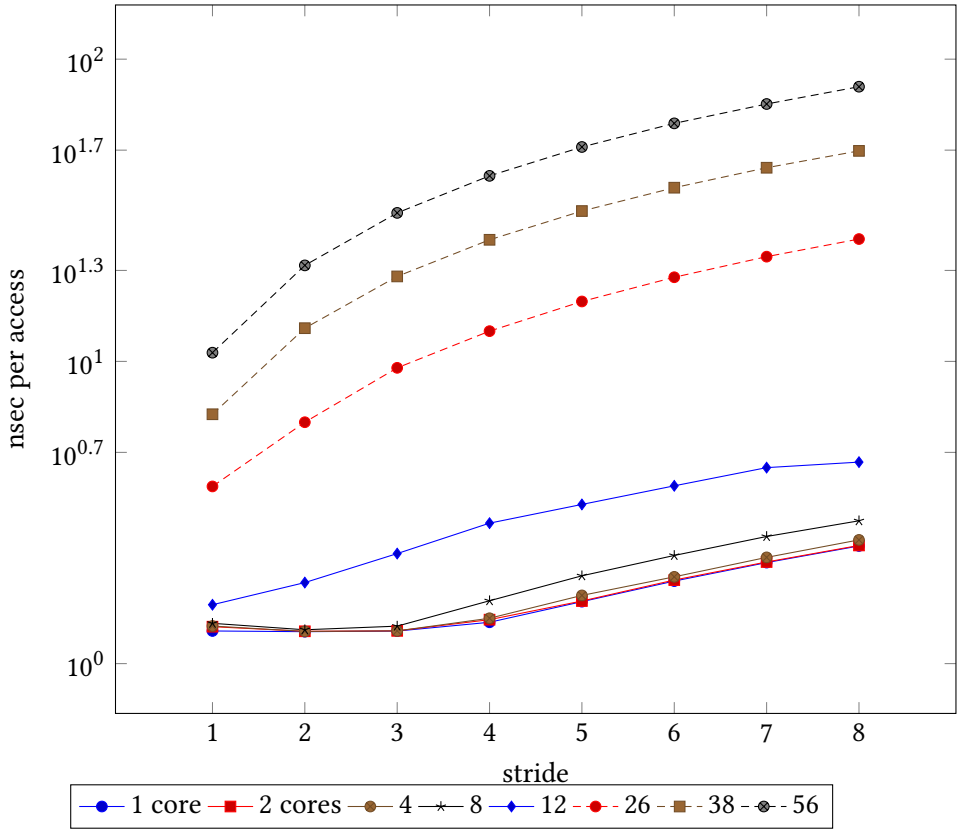


Figure 1.20: Nano second per access for various core and stride counts.

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 1.22 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.

1.7.7 Cache associativity

There are many algorithms that work by recursive division of a problem, for instance the *Fast Fourier Transform (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.

In section 1.3.4.9 you saw how the operation of adding a small number of vectors

$$\forall_j : y_j = y_j + \sum_{i=1}^{m} x_{i,j}$$

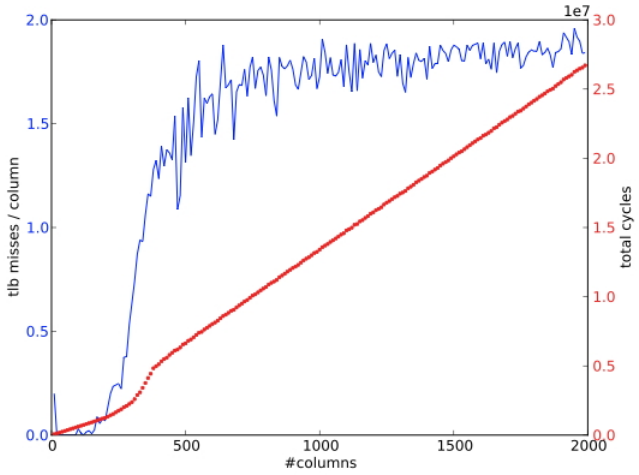


Figure 1.21: Number of TLB misses per column as function of the number of columns; columnwise traversal of the array.

is a problem for direct mapped caches or set-associative caches with associativity.

As an example we take the *AMD Opteron*, which has an L1 cache of 64K bytes, and which is two-way set associative. Because of the set associativity, the cache can handle two addresses being mapped to the same cache location, but not three or more. Thus, we let the vectors be of size $n = 4096$ doubles, and we measure the effect in cache misses and cycles of letting $m = 1, 2, \ldots$.

First of all, we note that we use the vectors sequentially, so, with a cacheline of eight doubles, we should ideally see a cache miss rate of $1/8$ times the number of vectors m. Instead, in figure 1.23 we see a rate approximately proportional to m, meaning that indeed cache lines are evicted immediately. The exception here is the case $m = 1$, where the two-way associativity allows the cachelines of two vectors to stay in cache.

Compare this to figure 1.24, where we used a slightly longer vector length, so that locations with the same j are no longer mapped to the same cache location. As a result, we see a cache miss rate around $1/8$, and a smaller number of cycles, corresponding to a complete reuse of the cache lines.

Two remarks: the cache miss numbers are in fact lower than the theory predicts, since the processor will use prefetch streams. Secondly, in figure 1.24 we see a decreasing time with increasing m; this is probably due to a progressively more favorable balance between load and store operations. Store operations are more expensive than loads, for various reasons.

1.7.8 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.

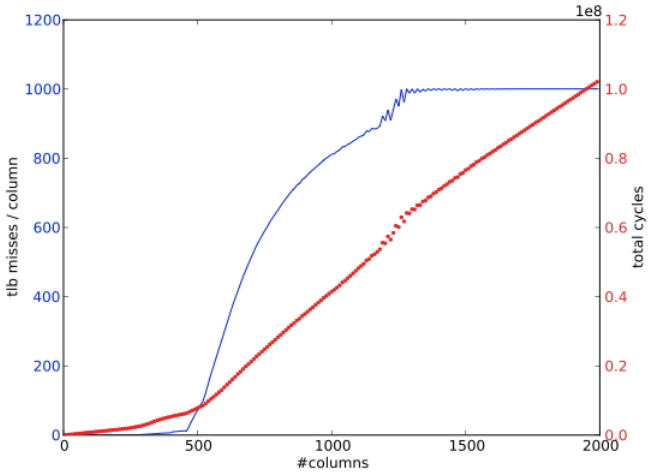


Figure 1.22: Number of TLB misses per column as function of the number of columns; rowwise traversal of the array.

Exercise 1.21. 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 1.22. 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 19.5.2*.

1.7.9 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*

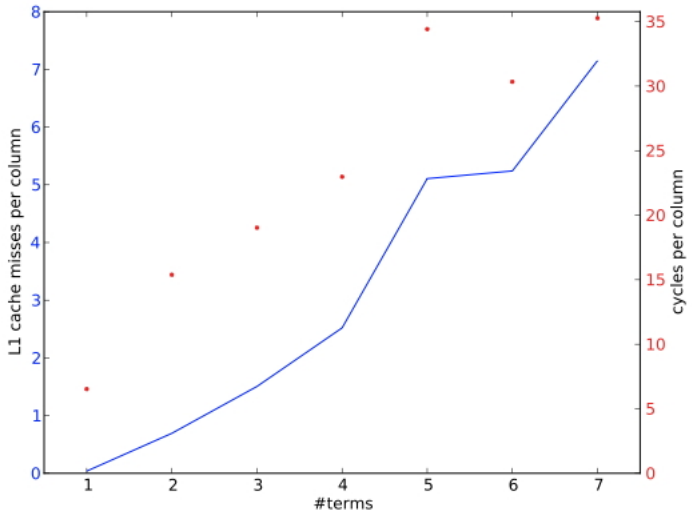


Figure 1.23: The number of L1 cache misses and the number of cycles for each j column accumulation, vector length 4096.

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 */
```

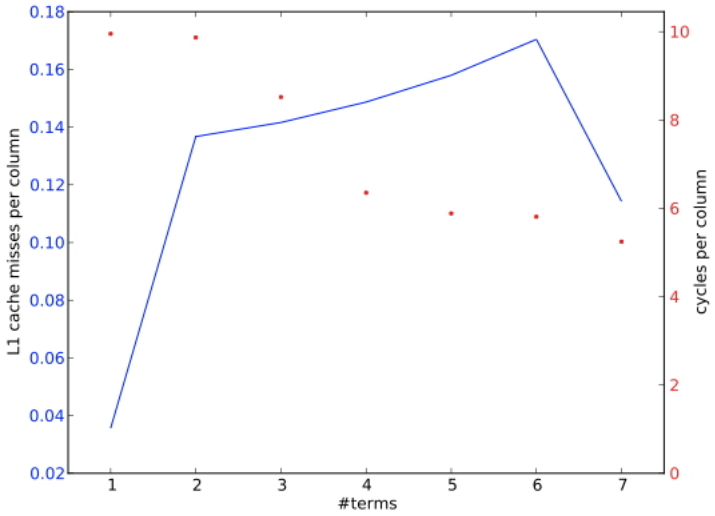


Figure 1.24: The number of L1 cache misses and the number of cycles for each j column accumulation, vector length 4096 + 8.

```
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 1.23. 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:

```
// 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:

```
// 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.

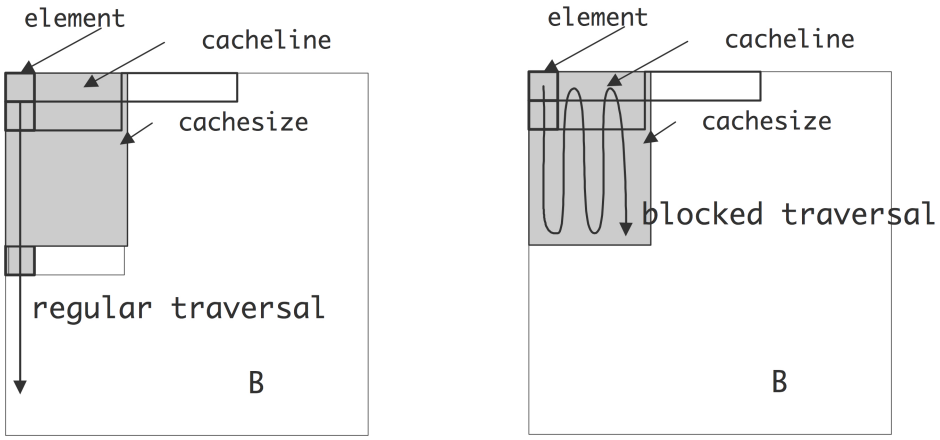


Figure 1.25: Regular and blocked traversal of a matrix.

Figure 1.25 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
```

1.7.10 Gradual underflow

The way a processor handles underflow can have an effect on the performance of a code. See section 3.8.1.

1.7.11 Optimization strategies

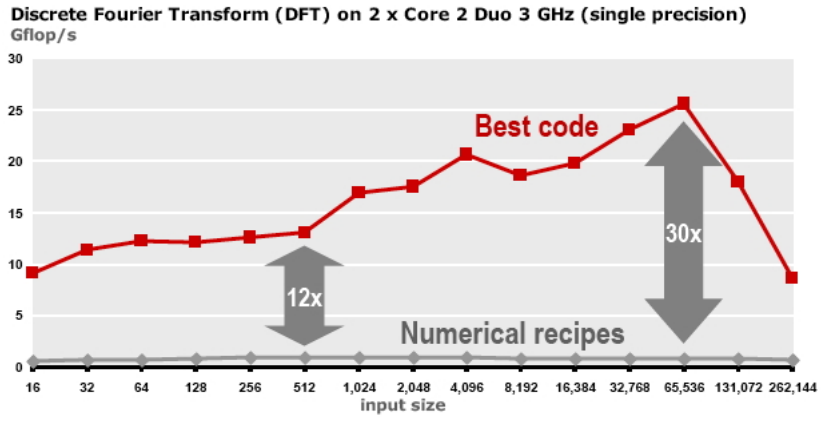


Figure 1.26: Performance of naive and optimized implementations of the Discrete Fourier Transform.

Figures 1.26 and 1.27 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:

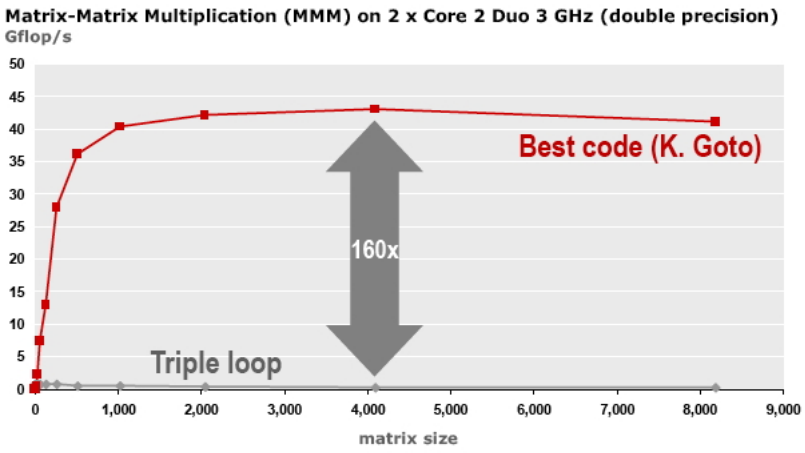


Figure 1.27: Performance of naive and optimized implementations of the matrix-matrix product.

- Compilers are not able to extract anywhere close to optimal performance[4] .
- 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 [188] for Blas kernels, and Spiral [162] for transforms.

1.7.12 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[5] , 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:

4. 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.
5. We are conveniently ignoring matters of set-associativity here, and basically assuming a fully associative cache.

```
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 1.7.4 for an elaborate discussion.)

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*. This is typically done by using a *divide-and-conquer* strategy, that is, recursive subdivision of a problem.

As 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.

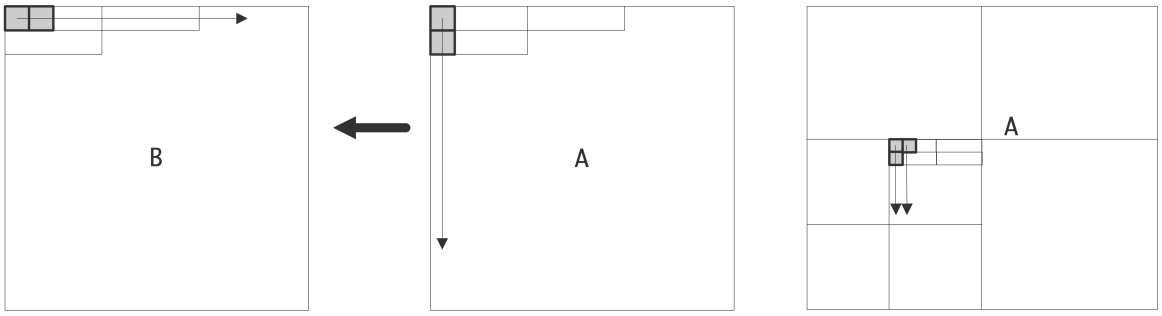


Figure 1.28: Matrix transpose operation, with simple and recursive traversal of the source matrix.

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 1.28. 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 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 [82].

See section 6.8.4 for a discussion of such techniques in stencil computations.

1.7.13 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.

1.8 Further topics

1.8.1 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.

1.8.1.1 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
2014	6,000,000,000	18-core Haswell
2017	20,000,000,000	32-core AMD Epyc
2019	40,000,000,000	64-core AMD Rome

Chronology of Moore's Law (courtesy Jukka Suomela, 'Programming Parallel Computers')

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

Charge	$q = CV$	
Work	$W = qV = CV^2$	(1.1)
Power	$W/\text{time} = WF = CV^2F$	

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

1.8.1.2 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 1.29 gives a dramatic illustration of the heat that a chip would give off, if single-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

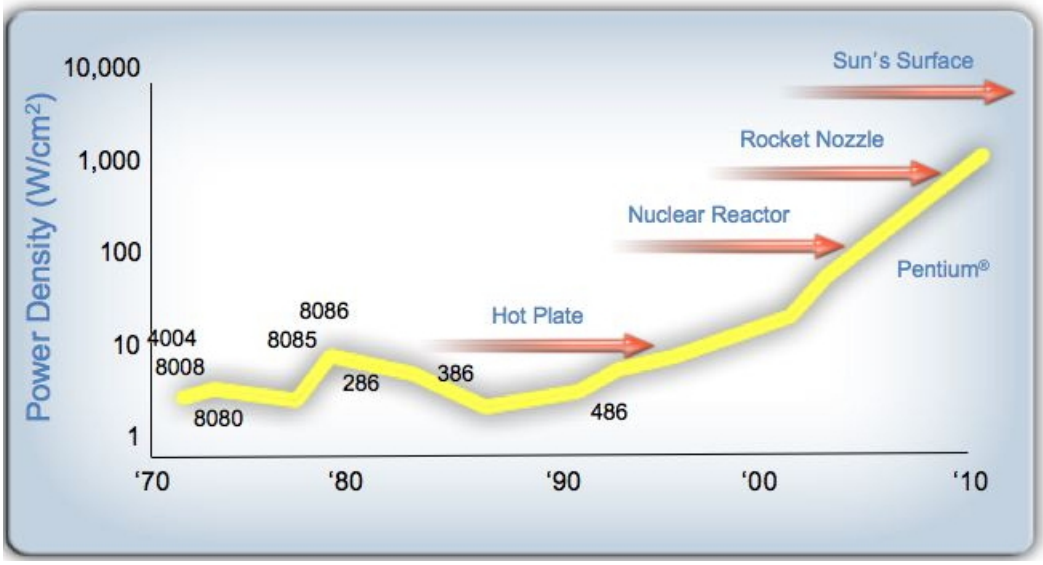


Figure 1.29: Projected heat dissipation of a CPU if trends had continued – this graph courtesy Pat Helsinger.

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.

1.8.1.3 *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.

1.8.2 Operating system effects

HPC practitioners typically don't worry much about the *Operating System (OS)*|textbf. 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*.

1.9 Review questions

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

Exercise 1.24. True or false. The code

```
for (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 1.25. Give an example of a code fragment where a 3-way associative cache will have conflicts, but a 4-way cache will not.

Exercise 1.26. 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

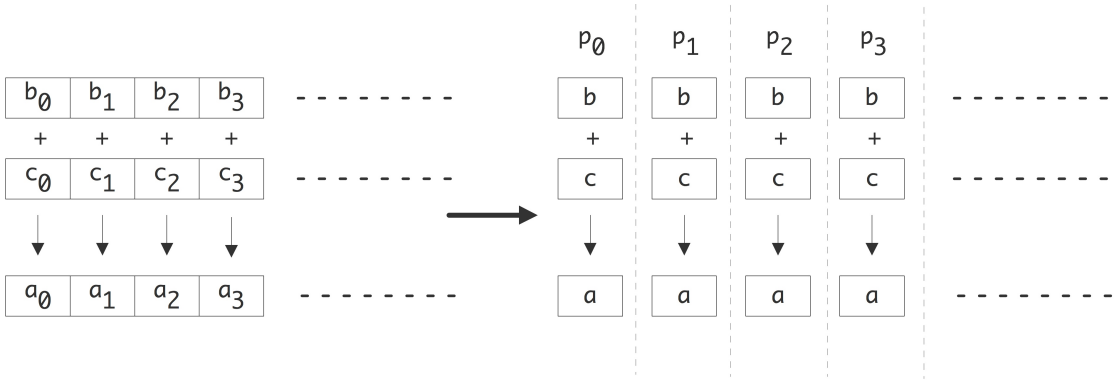


Figure 2.1: Parallelization of a vector addition.

```
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.

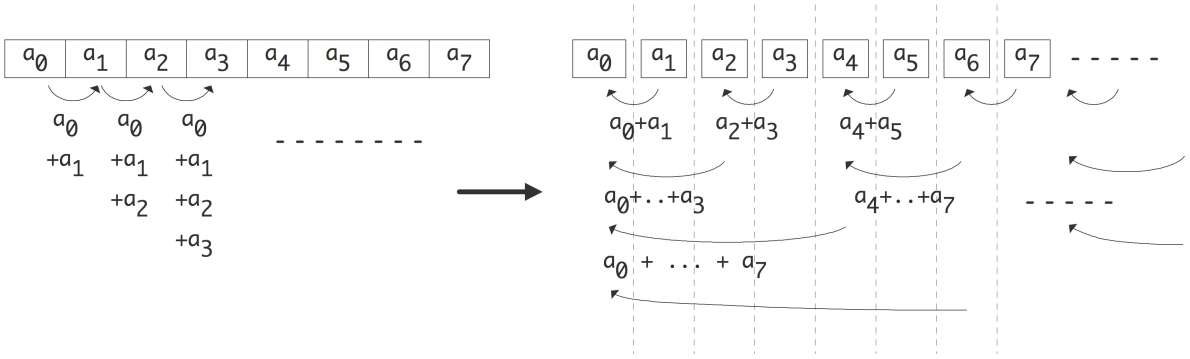


Figure 2.2: Parallelization of a vector reduction.

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 19) 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

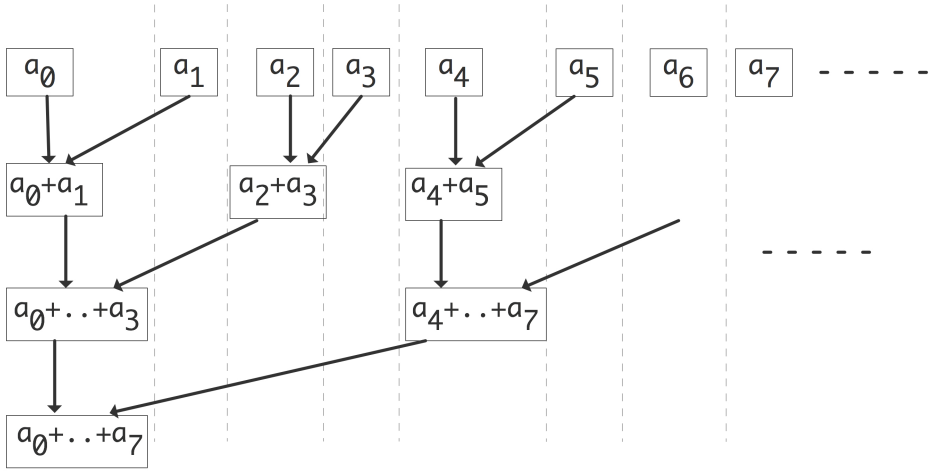


Figure 2.3: Communication structure of a parallel vector reduction.

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 6.10.2. 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 6.10.1. In general, the whole of chapter 6 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, $0 < E_p \leq 1$.

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 2.6. 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

$$T_o = pT_p - T1.$$

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 8.6) 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 6.2.2 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 2.7. 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 2.8. 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 2.9. 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* [4], which we will now formulate.

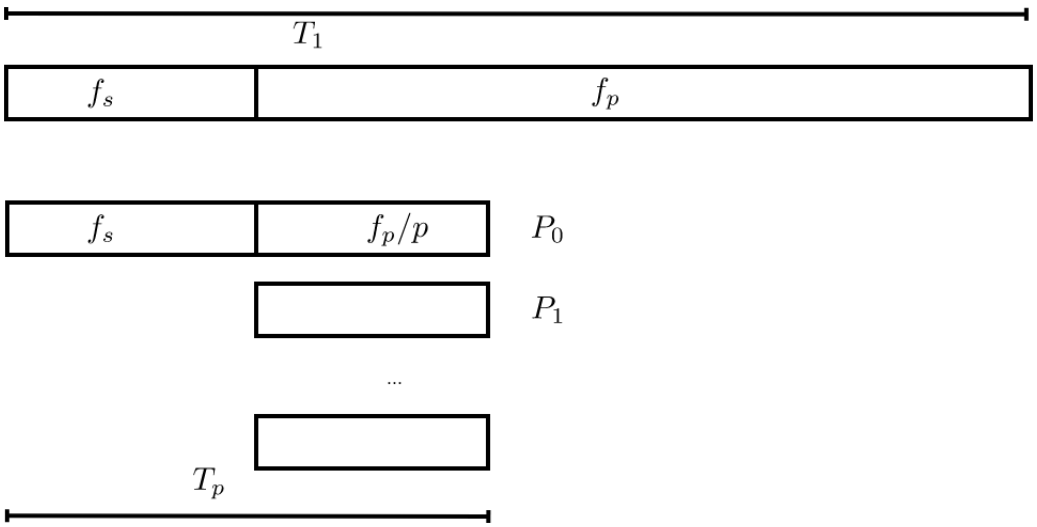


Figure 2.4: Sequential and parallel time in Amdahl's analysis.

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 2.10. 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 2.11. 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 [130, 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.) This is:

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

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:

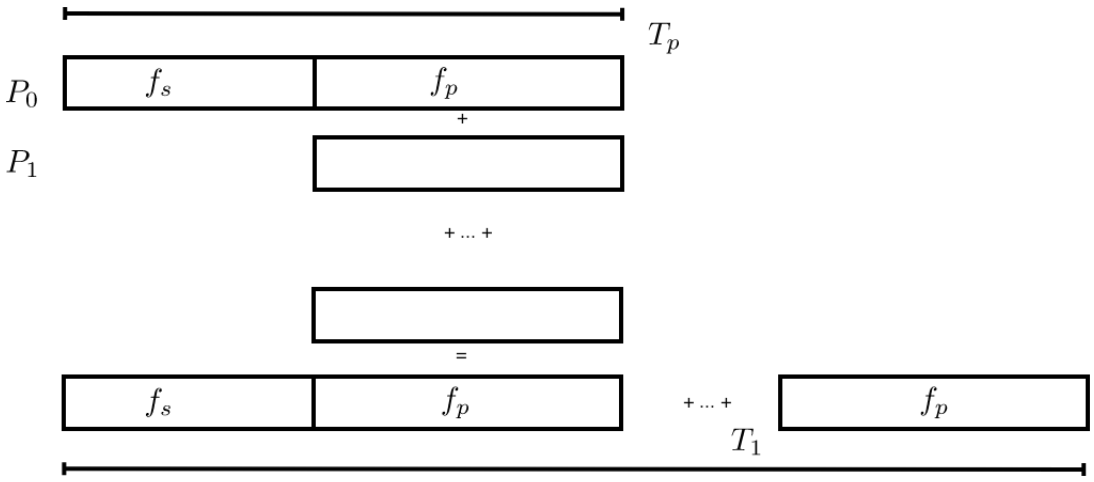


Figure 2.5: Sequential and parallel time in Gustafson's analysis.

- 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 2.12. 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 6.2.2 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 2.13. 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.

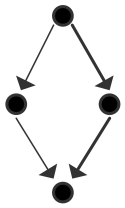
To make these notions precise, we define the following concepts:

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* $T_p = T_\infty$

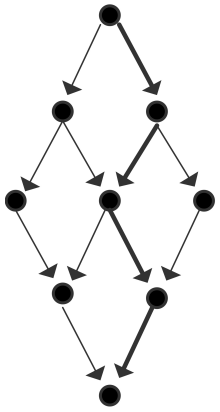
With these concepts, we can define the *average parallelism* of an algorithm as T_1/T_∞, and the length of the critical path is T_∞.

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.



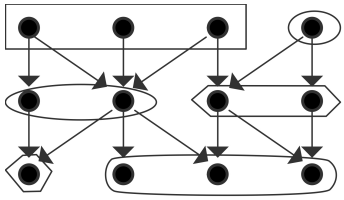
The maximum number of processors that can be used is 2 and the average parallelism is 4/3:

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



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

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



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 $\equiv 1$:

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

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 2.14. 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?

Exercise 2.15. Apply Brent's theorem to Gaussian elimination, assuming that add/multiply/-division all take one unit time.
Describe the critical path and give the length. What is the resulting upper bound on the parallel runtime?
How many processors could you theoretically use? What speedup and efficiency does that give?

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 2.16. 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 2.17. 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 2.18. 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 8.6) 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 7 *A historical anecdote.*

```
Message:  1023110, 88 lines Posted:  5:34pm EST, Mon Nov
25/85, imported:  ....  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" [12].

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:

$$M = Pm \qquad \text{total memory.}$$

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.

Exercise 2.19. Explore simulation scaling in the context of the *Linpack benchmark*, that is, Gaussian elimination. Ignore the system solving part and only consider the factorization part; assume that it can be perfectly parallelized.

1. Suppose you have a single core machine, and your benchmark run takes time T with M words of memory. Now 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?
2. Now suppose you have a machine with P processors, each with M memory, and your benchmark run takes time T. You buy a machine with $2P$ processors, of the same clock speed and core count, and you want to do a benchmark run, again taking time T. How much memory does each node take?

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 [150].

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.

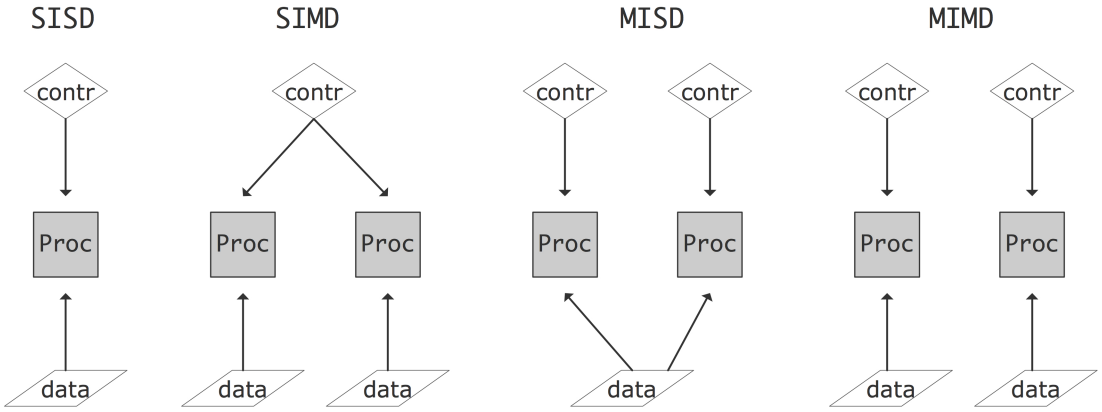


Figure 2.6: The four classes of the Flynn's taxonomy.

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):

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 [112] in the 1980s, and *Goodyear* built a 16K processor MPP [10] in the 1970s.

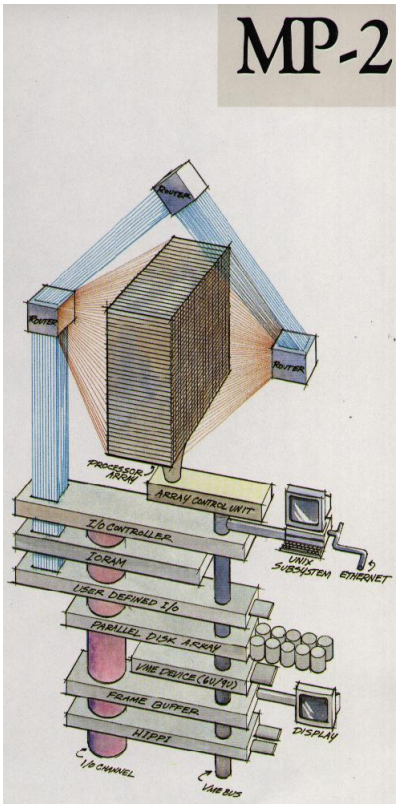


Figure 2.7: Architecture of the MasPar 2 array processors.

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 [167], 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 1.3.3), vector operations use *vector registers*. The locations in a vector register are sometimes referred to as *SIMD lanes*.

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

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 1.3.4.7), 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.

2.3.2 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*

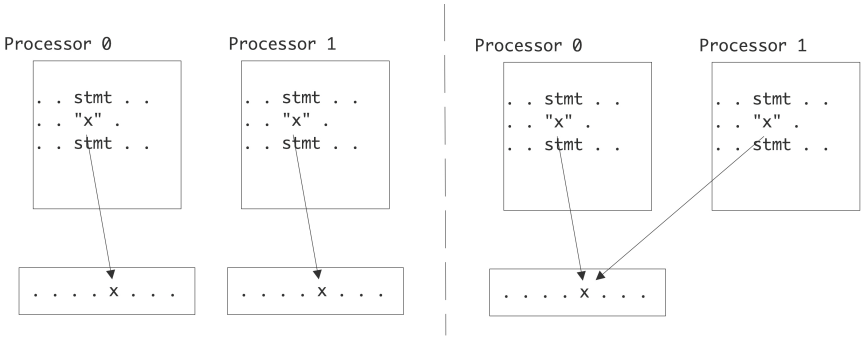


Figure 2.8: References to identically named variables in the distributed and shared memory case.

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 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

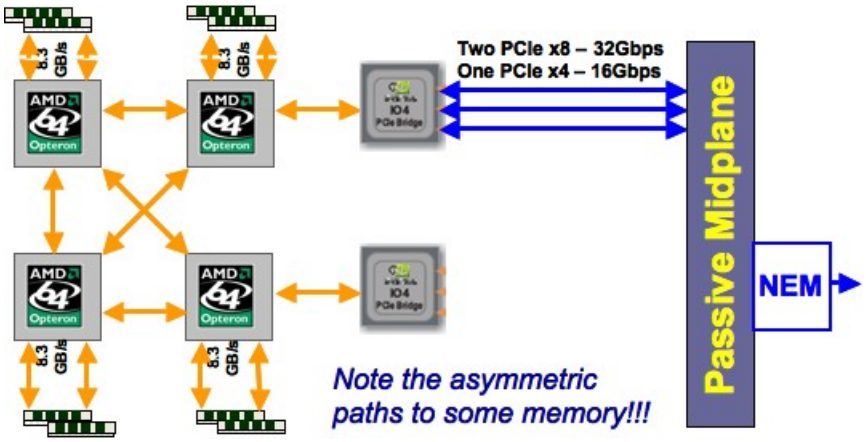


Figure 2.9: Non-uniform memory access in a four-socket motherboard.

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.

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 ← shiftright(b)
bright ← shiftleft(b)
a ← (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.

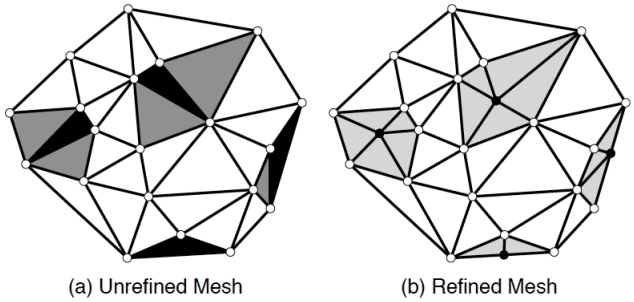


Figure 2.10: A mesh before and after refinement.

In pseudo-code, this can be implemented as in figure 2.11. (This figure and code are to be found in [124], 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.

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*.

```
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.

2.5.5 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 6.3.2.

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*

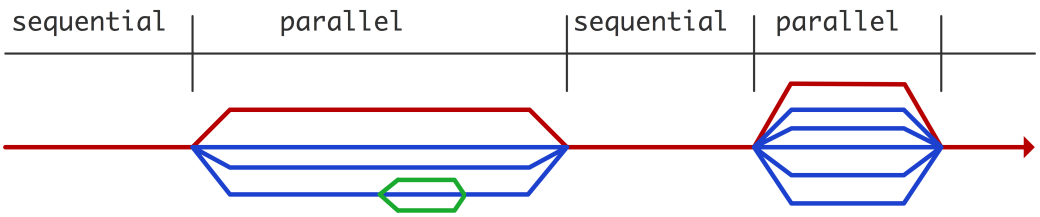


Figure 2.12: Thread creation and deletion during parallel execution.

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 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.	
		$\texttt{I} = 0$			
read $\texttt{I} = 0$	read $\texttt{I} = 0$	read $\texttt{I} = 0$	read $\texttt{I} = 0$	read $\texttt{I} = 0$	
compute $\texttt{I} = 2$	compute $\texttt{I} = 3$	compute $\texttt{I} = 2$	compute $\texttt{I} = 3$	compute $\texttt{I} = 2$	
write $\texttt{I} = 2$			write $\texttt{I} = 3$	write $\texttt{I} = 2$	
	write $\texttt{I} = 3$	write $\texttt{I} = 2$			read $\texttt{I} = 2$
					compute $\texttt{I} = 5$
					write $\texttt{I} = 5$
$\texttt{I} = 3$		$\texttt{I} = 2$		$\texttt{I} = 5$	

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 nondeterministic, 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 [3]. 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 [121].)

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 [128].

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

- 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?

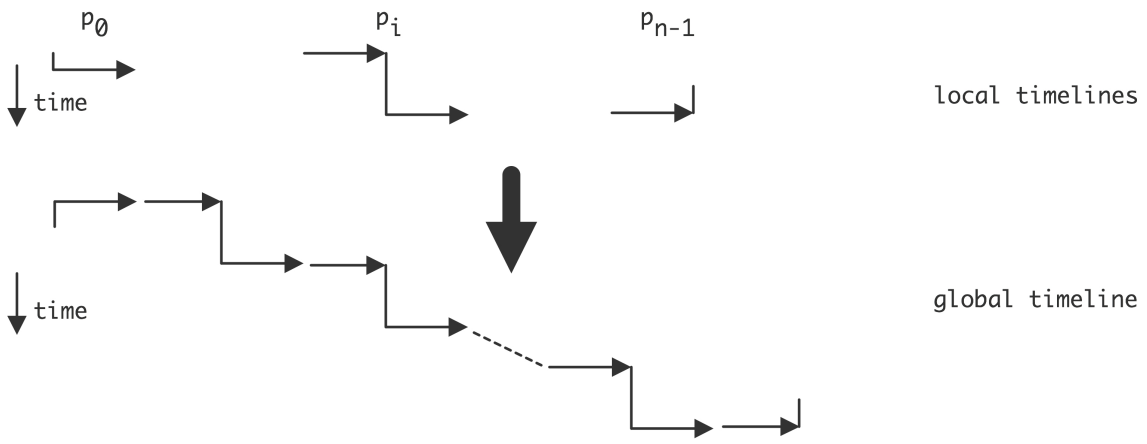


Figure 2.15: Local and resulting global view of an algorithm for sending data to the right.

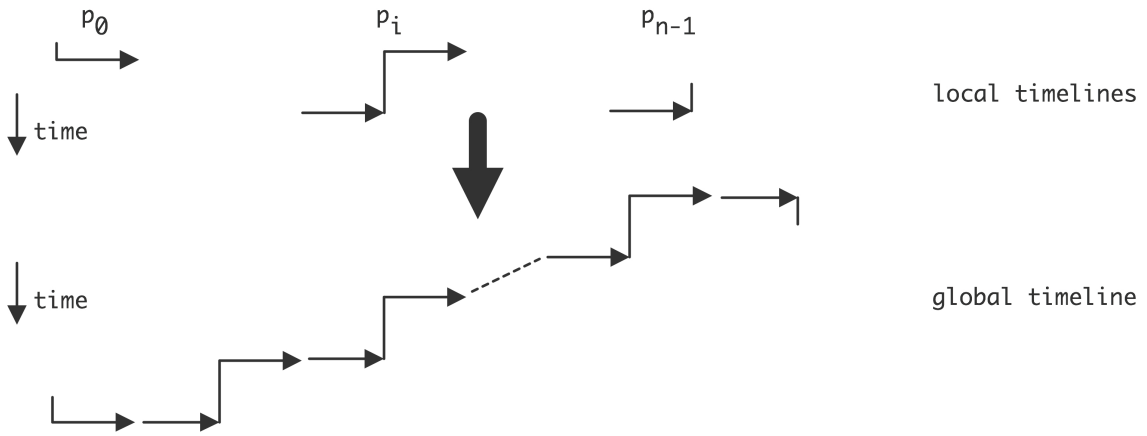


Figure 2.16: Local and resulting global view of an algorithm for sending data to the right.

- 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*.

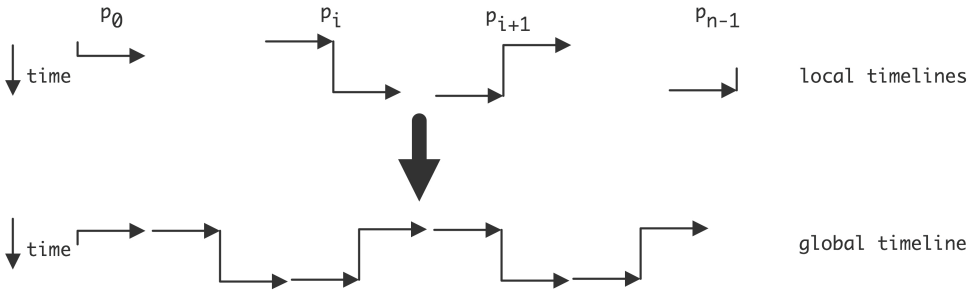


Figure 2.17: Local and resulting global view of an algorithm for sending data to the right.

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:

- 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) [173] 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[3]. 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:

3. This 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].

```
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 6.1.

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 [154] 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 8 *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.

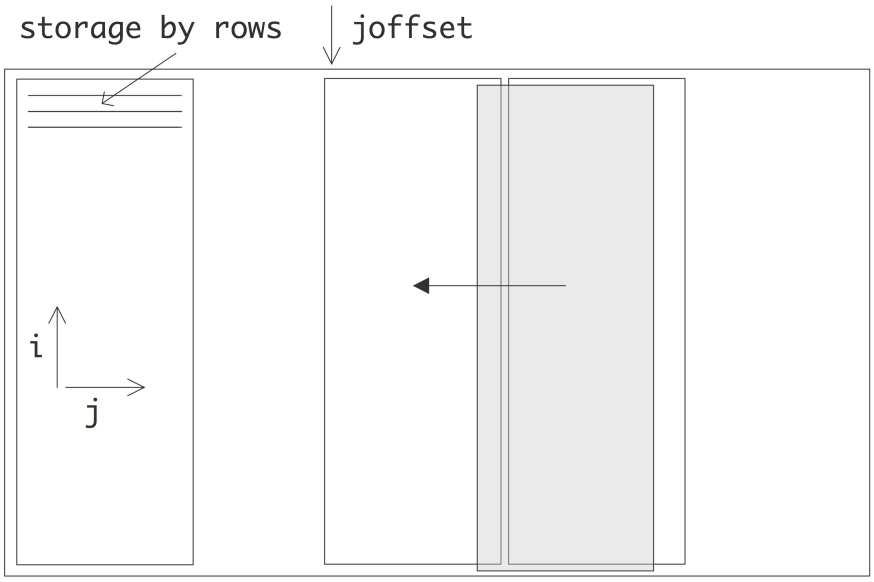


Figure 2.18: Data shift that requires communication.

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) [179] 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[4] (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,*]
```

4. This section quoted from Wikipedia

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[5] 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 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[6] 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

5. This section quoted from the Chapel homepage.
6. This section quoted from the Fortress homepage.

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 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 *ghost 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[7] , distributed over a processor grid of the same or lower dimension. Through library calls, 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 [119] 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++*[115].

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:

7. 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.

```
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 [180, 172]. 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* [143].

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 6.1.2.

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. 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 6.2.1). 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.

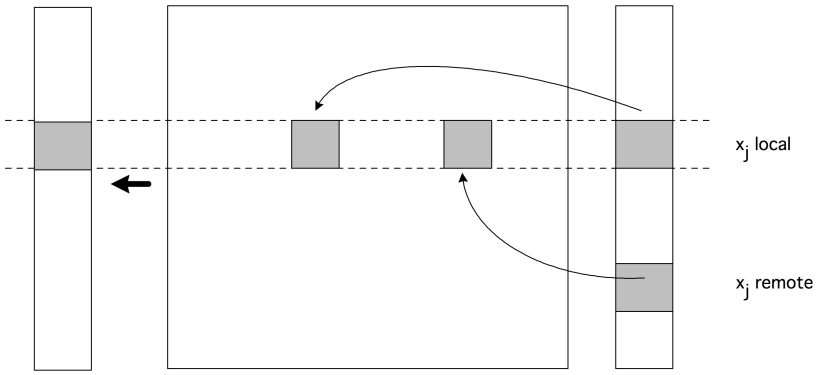


Figure 2.19: The parallel matrix-vector product with a blockrow distribution.

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 9 *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

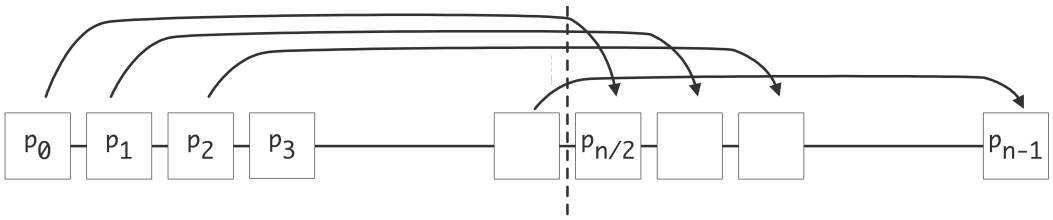


Figure 2.20: Contention for a network link due to simultaneous messages.

the wire between $p_{n/2-1}$ and $p_{n/2}$. This sort of conflict is called *congestion* or *contention*. Clearly, the more 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.

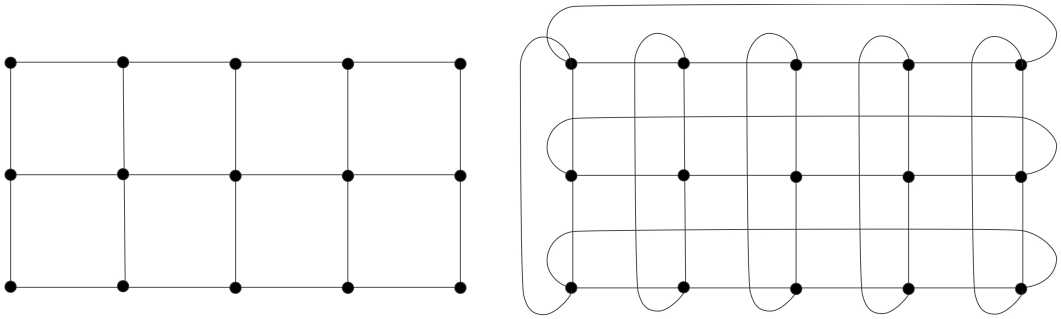


Figure 2.21: A 2D grid with torus connections.

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.

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 its location in the cube; see figure 2.22.

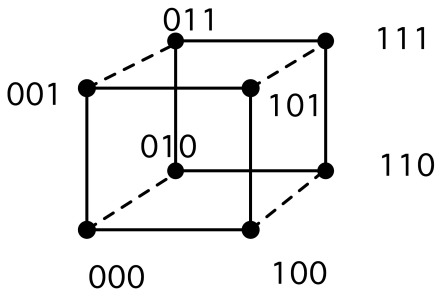


Figure 2.22: Numbering of the nodes of a hypercube.

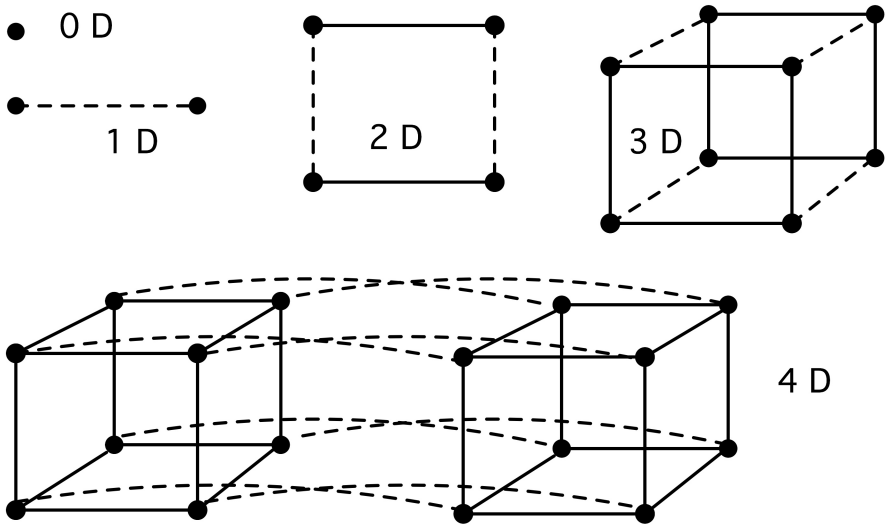


Figure 2.23: Hypercubes.

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				
3D Gray code:	2D code and reflection:	0	1	1	0	⋮	0	1	1	0
		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.

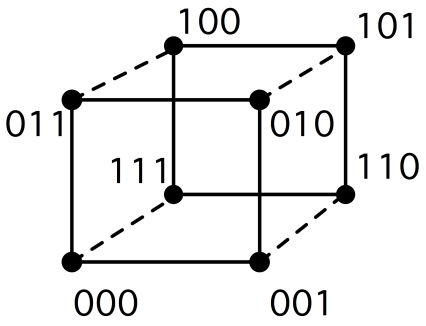


Figure 2.25: Gray code numbering of the nodes of a hypercube.

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.

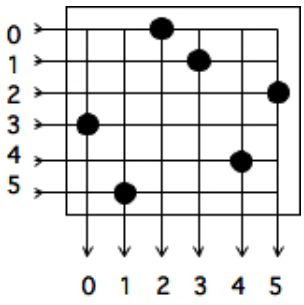


Figure 2.26: A simple cross bar connecting 6 inputs to 6 outputs.

2.7.6.2 Butterfly exchange

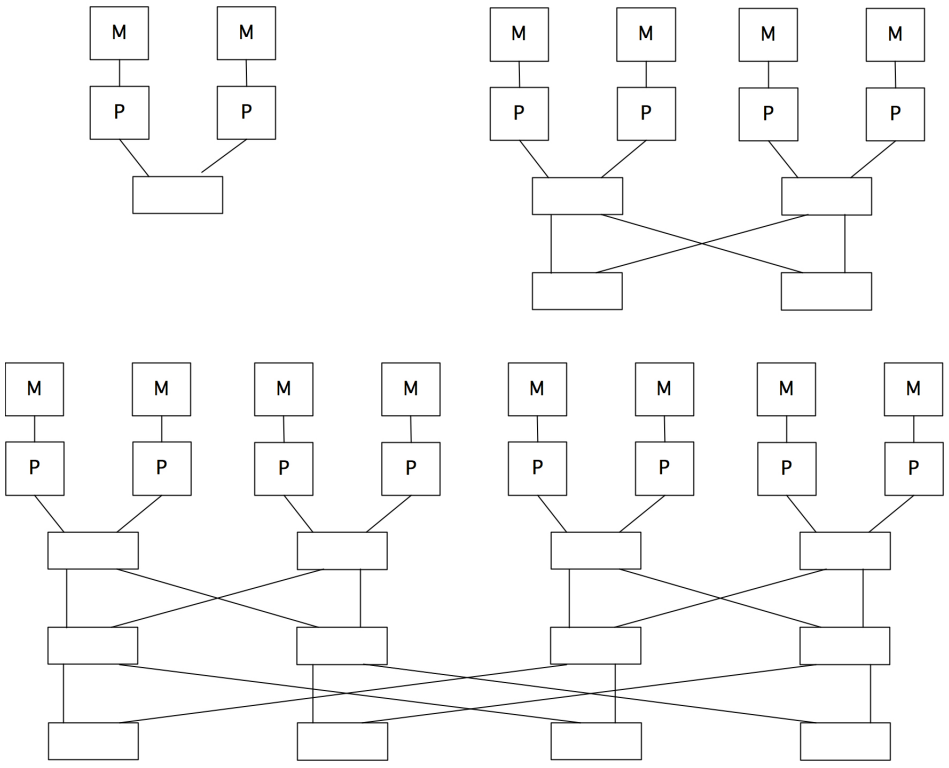


Figure 2.27: Butterfly exchange networks for 2,4,8 processors, each with a local memory.

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.

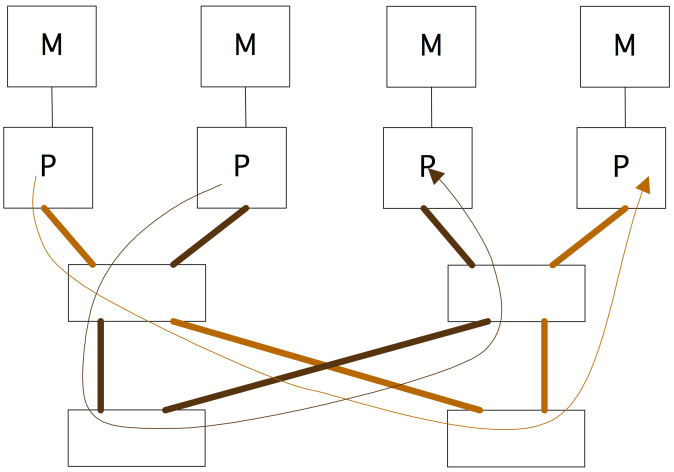


Figure 2.28: Two independent routes through a butterfly exchange network.

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. 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

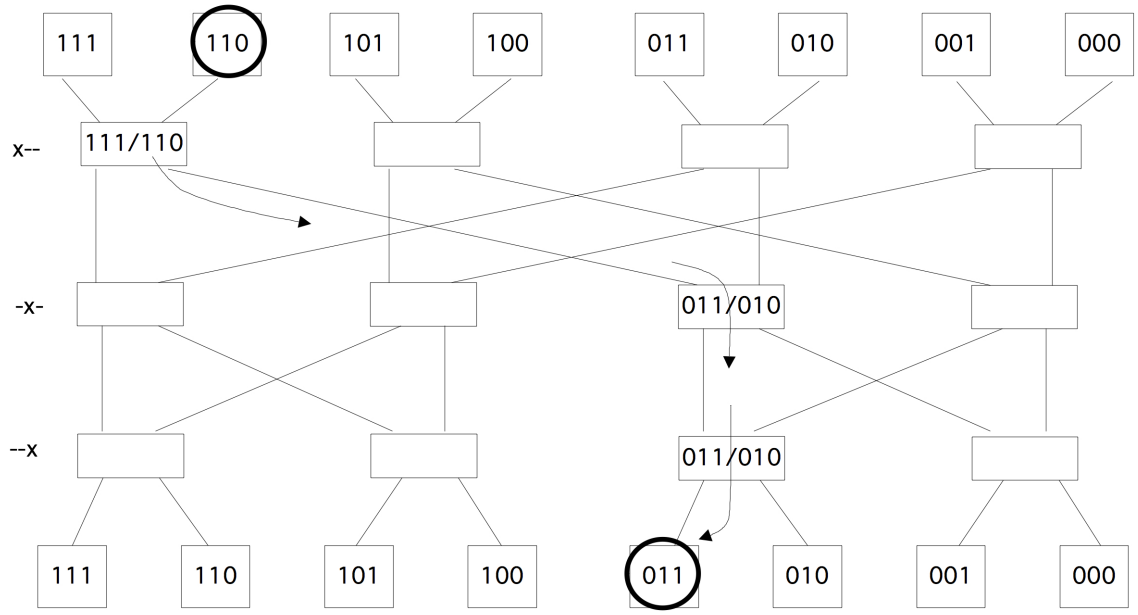


Figure 2.29: Routing through a three-stage butterfly exchange network.

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 [135] 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 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 switch*es (that is, cabinents housing the top levels of the fat tree), connecting 91 processor cabinets.

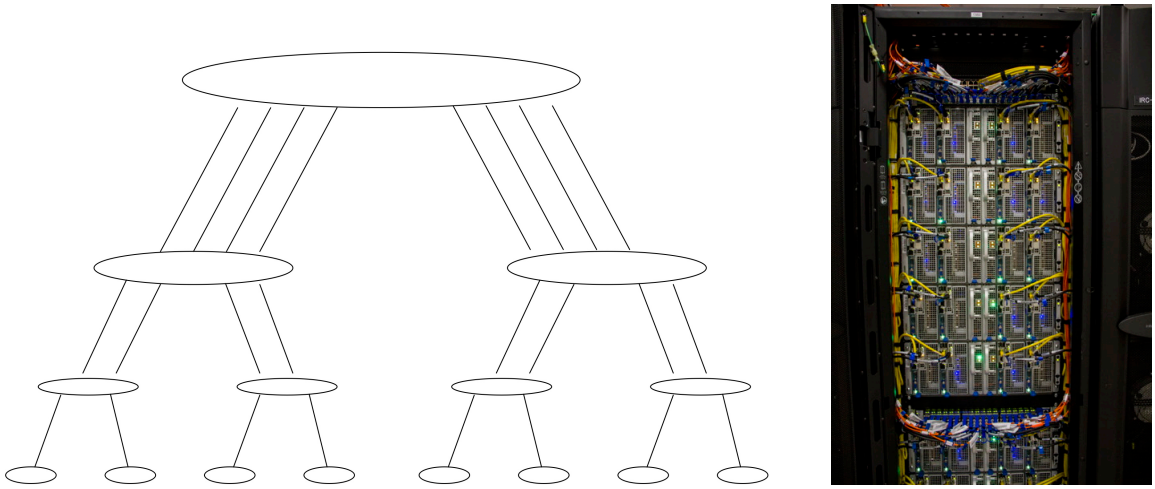


Figure 2.30: A fat tree with a three-level interconnect (left); the leaf switches in a cabinet of the Stampede cluster (right).

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

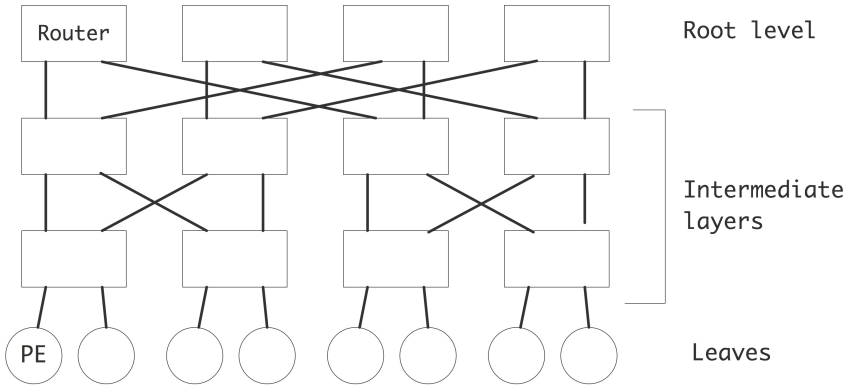


Figure 2.31: A fat-tree built from simple switching elements.

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 card*s.)

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.

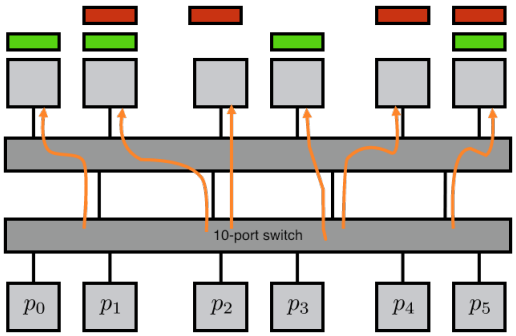


Figure 2.32: Port contention in an over-subscribed switch.

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, 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 $w' = 2$.

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 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.

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.

Figure 2.33: Networks switches for the TACC Ranger and Stampede clusters.

- 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.

Figure 2.34: A BlueGene computer.

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 6.1. 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*[8].

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 80287, added to it. This 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.

8. Tera Computer renamed itself *Cray Inc.* after acquiring *Cray Research* from *SGI*.

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:

- 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.

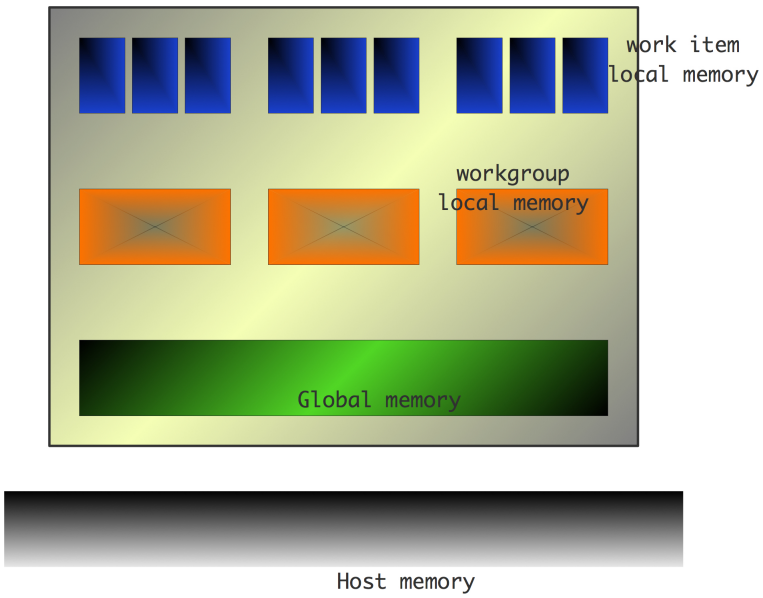


Figure 2.35: Memory structure of a GPU.

2.9.3.1 *SIMD-type programming with kernels*

Present day GPUs[9] 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

9. The most popular GPUs today are made by NVidia, and are programmed in *CUDA*, an extension of the C language.

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.

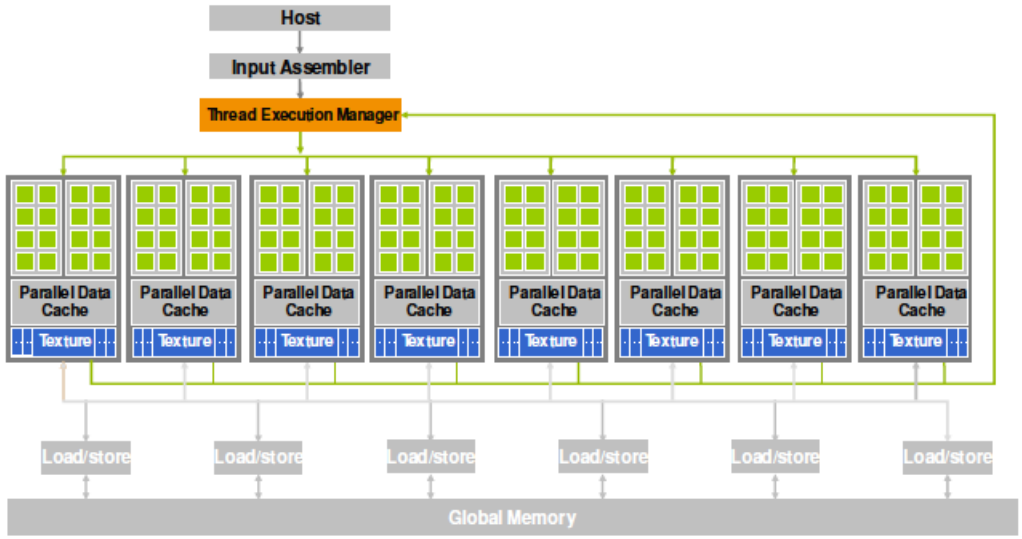


Figure 2.36: Diagram of a GPU.

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)
```

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.

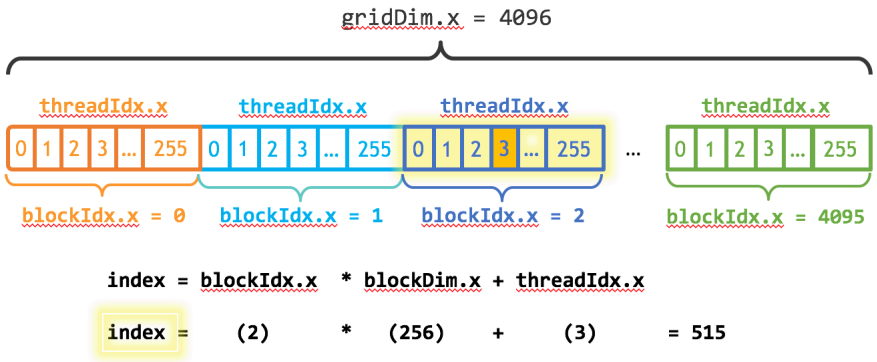


Figure 2.37: Thread indexing in CUDA.

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 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 increas-

ing, 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 6.5 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 6.2 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

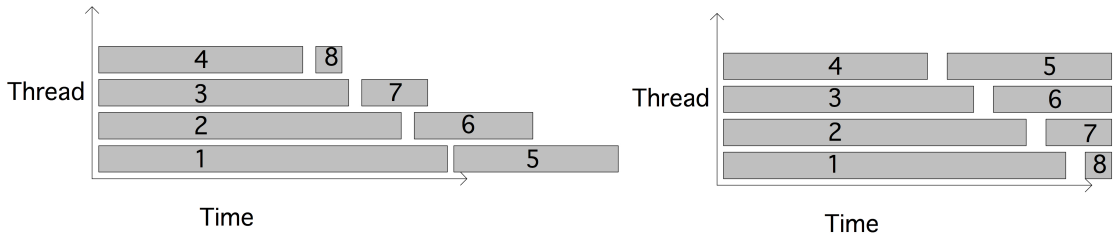


Figure 2.38: Static or round-robin (left) vs dynamic (right) thread scheduling; the task numbers are indicated.

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 6.12.

There are results that show that randomized assignment of tasks to processors is statistically close to optimal [117], 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 [160].)

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 19 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 19.6): 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 6.5.3. 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[10] . This is illustrated in figure 2.39. If we keep track 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.

- 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,

10. For a detailed discussion, see [28].

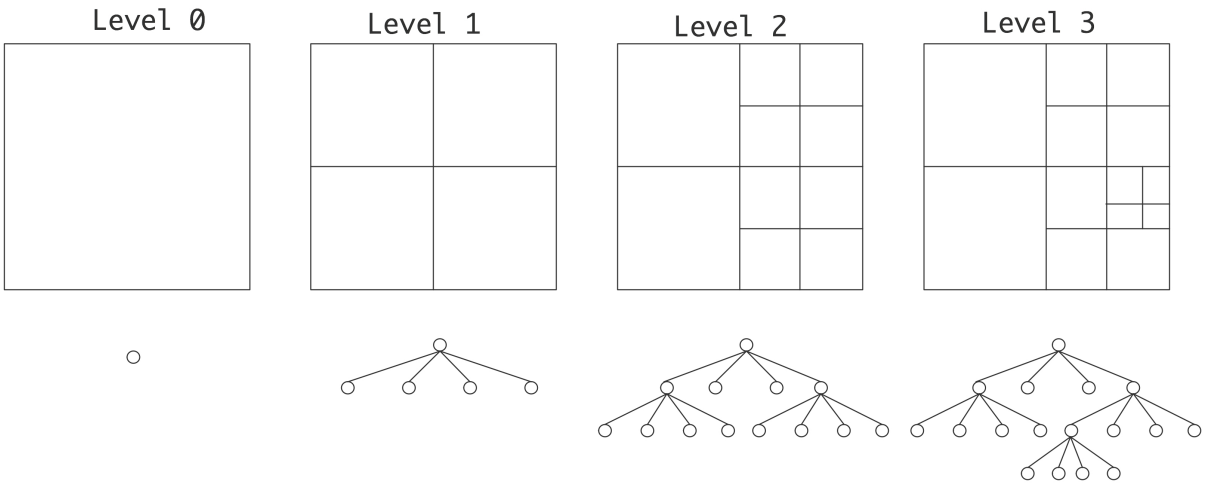


Figure 2.39: Adaptive refinement of a domain in subsequent levels.

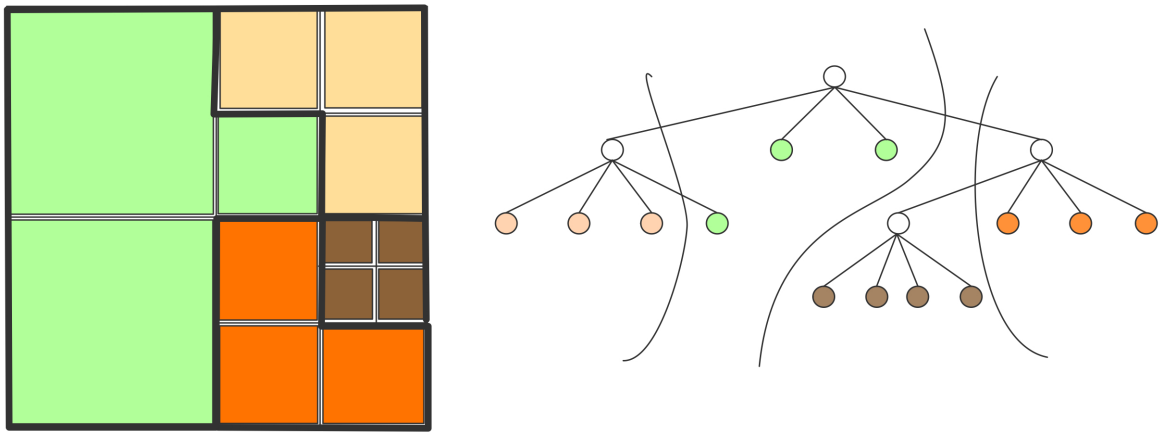


Figure 2.40: Load distribution of an adaptively refined domain.

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 2.42 stand for a definition: a SFC is a recursively defined curve that touches each subdomain once[11] . The SFC has the property that domain elements that are close together physically will be close together on the curve, so if we map the SFC to a linear ordering of processors we will preserve the locality of the

11. 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 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

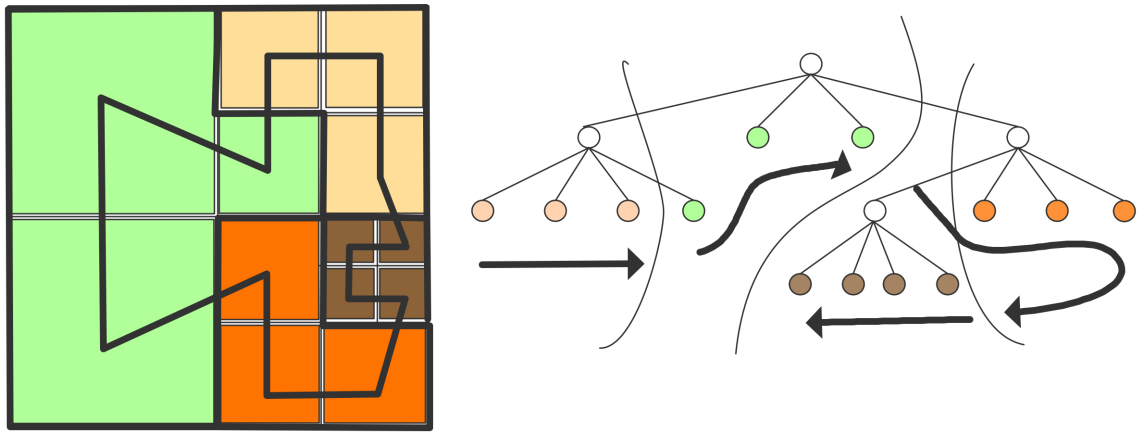


Figure 2.41: A space filling curve for the load balanced tree.

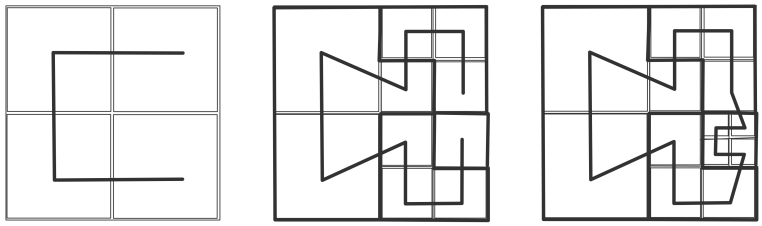


Figure 2.42: Space filling curves, regularly and irregularly refined.

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 [187] and [176].)

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)

Brouwer.

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 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[12] .

- 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[168]. Here large central datasets are being made available to astronomers as in

12. Based on a blog post by Ricky Ho: http://blogs.globallogic.com/five-cloud-computing-patterns.

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[13] 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

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*.

13. 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.

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 9.4) 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 8.5.1.

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 6.5.6.

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*. *Linpack* is a package for linear algebra operations, and no longer in 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. It also has the property that the amount of work outweighs the amount of 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 (a *bandwidth-bound* algorithm).

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[14]. First of all, figure 2.43 shows the evolution of architecture types by

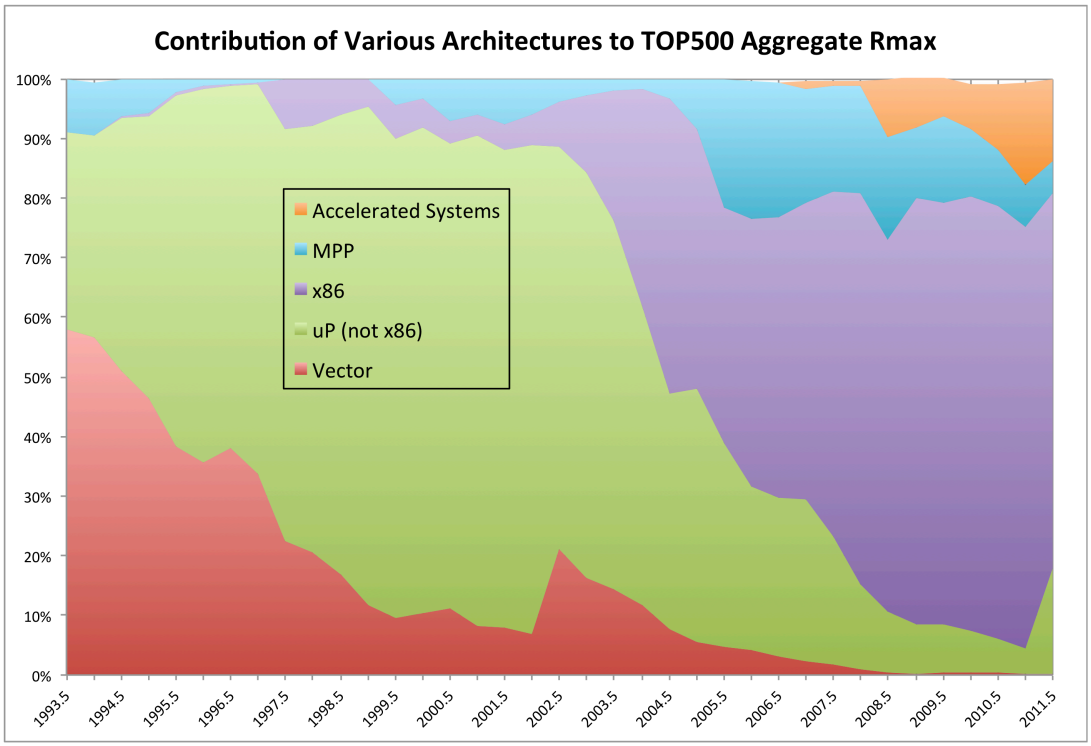


Figure 2.43: Evolution of the architecture types on the top500 list.

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.
- 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

14. The graphs contain John McCalpin's analysis of the top500 data.

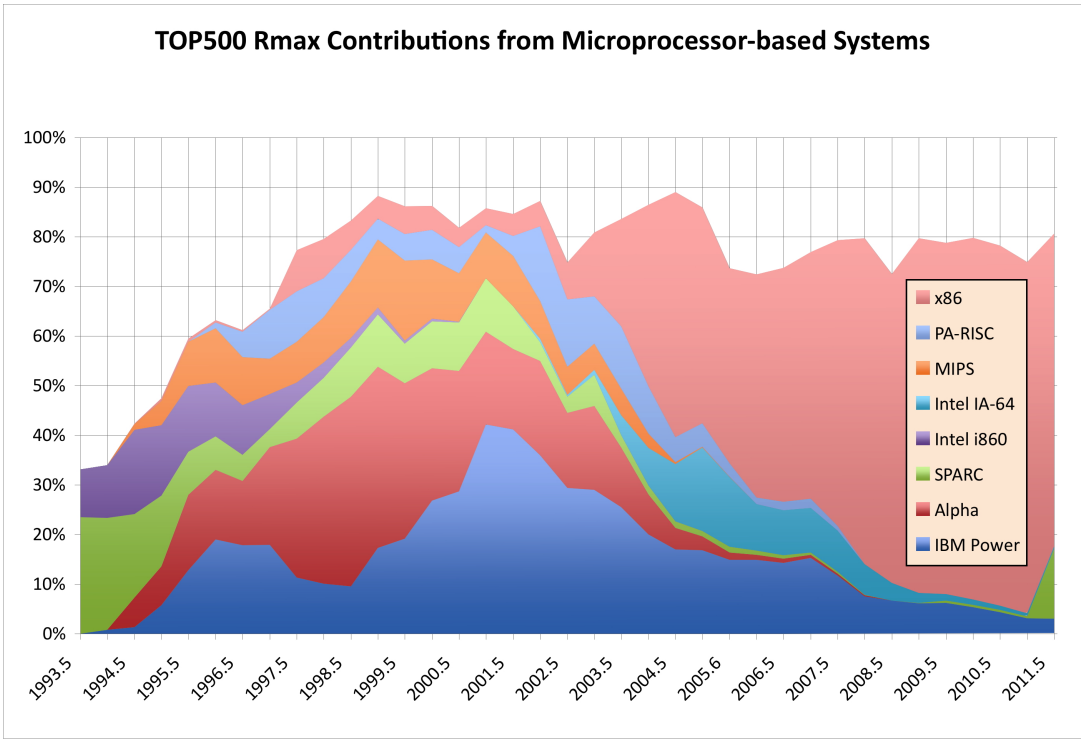


Figure 2.44: Evolution of the architecture types on the top500 list.

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.

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*.

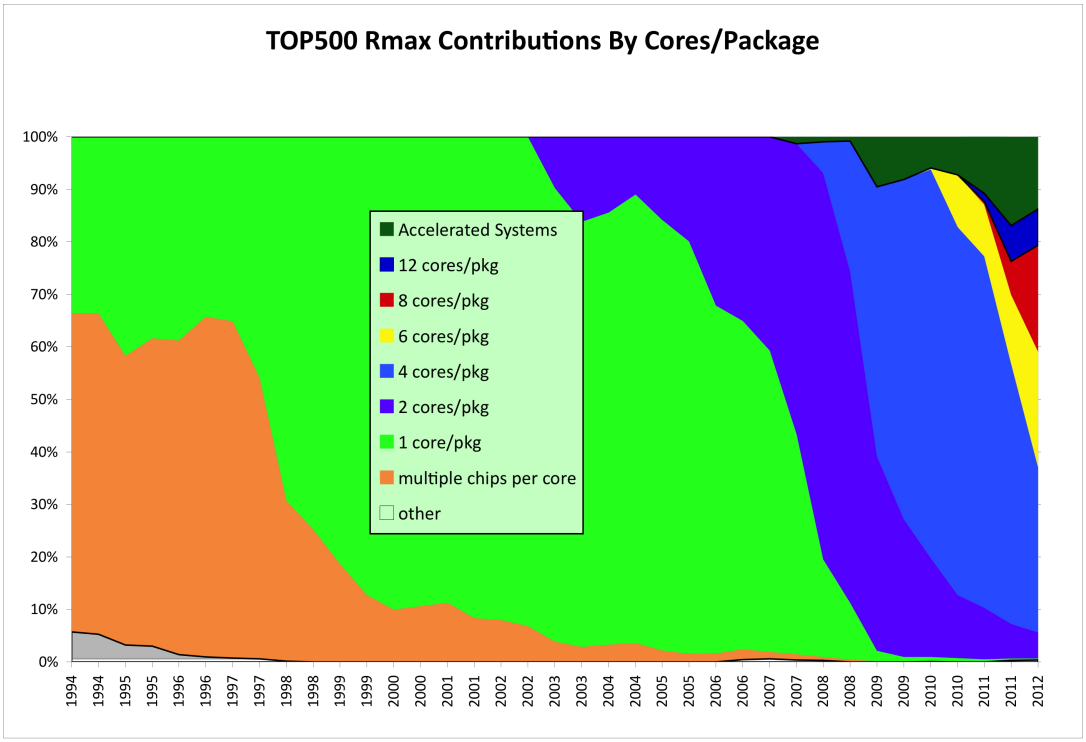


Figure 2.45: Evolution of the architecture types on the top500 list.

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.

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 [1];
- for secondary reading, see the book by Overton [155]; 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 [190].

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 like a sign bit; however note the above discussion.
- If you have a positive number n, you get $-n$ by flipping all the bits, and adding 1.

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.

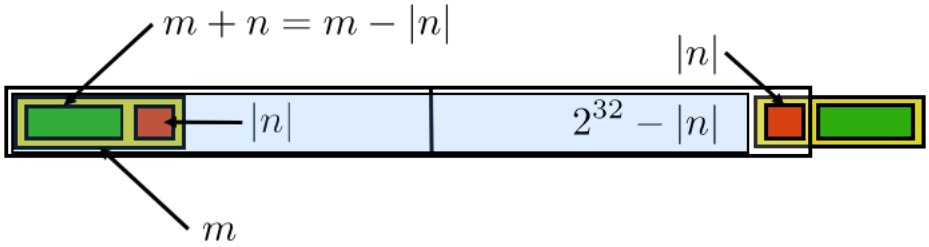


Figure 3.1: Addition with one positive and one negative number in 2's complement.

- 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* [1, 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 $0\ldots9$ 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 $0\ldots15$. More efficient encodings would encode $0\ldots999$ in ten bits, which could in principle store the numbers $0\ldots1023$. 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*.

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 mathematical 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 what is known as '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 the immediately following the first digit;
- 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, the bit pattern for the number zero is interpreted as $e = L$.

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 $1 \leq x_m < \beta$.

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. In the *IEEE 754* standard, this means that every 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.

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 in the following sections.

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	$\cdots$	$0,\beta-1$	
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	$\cdots$	0	
value	1	0	$\cdots$	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 IEEE standard arithmetic (section 3.4) 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] . At the time of this writing, only the IBM Power6 (and up) has hardware support for gradual underflow. Section 3.8.1 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 3.7. 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 3.4.1.1.)

In none of these cases will the computation end: the processor will continue, unless you tell it otherwise. The 'otherwise' consists of you telling the compiler to generate an *interrupt*, which halts the computation with an error message. See section 3.7.2.4.

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 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:

$$\begin{cases} x \in \left[\tilde{x}, \tilde{x} + \beta^{-t+1}\right) & \text{truncation} \\ x \in \left[\tilde{x} - \frac{1}{2}\beta^{-t+1}, \tilde{x} + \frac{1}{2}\beta^{-t+1}\right) & \text{rounding} \end{cases} \tag{3.22}$$

1. In real-time applications such as audio processing this phenomenon is especially noticeable; see http://phonophunk.com/articles/pentium4-denormalization.php?pg=3.

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 3.8. The number in this example had no exponent part. What are the error and relative error if there had been one?

Exercise 3.9. 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?)

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 3.10. 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 3.11. 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 3.12. 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} for those values. Use for instance the same 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*

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.

754 standard [1] 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 10 *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 11 *'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 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.

exponent	numerical value	range
$(0\cdots0)=0$	$\pm 0.s_1\cdots s_{23}\times 2^{-126}$	$s=0\cdots01\Rightarrow 2^{-23}\cdot 2^{-126}=2^{-149}\approx 10^{-50}$ $s=1\cdots11\Rightarrow (1-2^{-23})\cdot 2^{-126}$
$(0\cdots01)=1$	$\pm 1.s_1\cdots s_{23}\times 2^{-126}$	$s=0\cdots00\Rightarrow 1\cdot 2^{-126}\approx 10^{-37}$
$(0\cdots010)=2$	$\pm 1.s_1\cdots s_{23}\times 2^{-125}$	
...		
$(01111111)=127$	$\pm 1.s_1\cdots s_{23}\times 2^{0}$	$s=0\cdots00\Rightarrow 1\cdot 2^{0}=1$ $s=0\cdots01\Rightarrow 1+2^{-23}\cdot 2^{0}=1+\epsilon$ $s=1\cdots11\Rightarrow (2-2^{-23})\cdot 2^{0}=2-\epsilon$
$(10000000)=128$	$\pm 1.s_1\cdots s_{23}\times 2^{1}$	
...		
$(11111110)=254$	$\pm 1.s_1\cdots s_{23}\times 2^{127}$	
$(11111111)=255$	$s_1\cdots s_{23}=0\Rightarrow\pm\infty$ $s_1\cdots s_{23}\neq 0\Rightarrow$ `NaN`	

(3.30)

Figure 3.3: Interpretation of single precision numbers depending on the exponent bit pattern.

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 $d_1d_2\ldots d_t$ is interpreted as $1.d_1d_2\ldots d_t$.

The highest exponent is used to accommodate `Inf` and `Nan`; see section 3.4.1.1.

Exercise 3.13. 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(double 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 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 the sort of calculations in chapter 4, but not for others such as the *Lattice Boltzmann Method (LBM)*.

3.4.1 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 3.7.4.)

3.4.1.1 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` and `NaN` for 'infinity' and 'not a number'. 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 12 *NaN and Inf do not always propagate:*

- *A finite number divided by Inf is zero;*
- *The min/max of a finite number and NaN is not a NaN;*
- *Various transcendental functions treat Inf correctly, such as* $\tan^{-1}(\text{Inf}) = \pi/2$.

3.4.1.2 Divide by zero

Division by zero results in `Inf`.

3.4.1.3 Overflow

This exception is raised if a result is not representable as a finite number.

3.4.1.4 Underflow

This exception is raised if a number is too small to be represented.

3.4.1.5 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, the 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 3.14. 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].

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 3.8.4.

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].

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.

Exercise 3.15. 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 3.16. 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

$$\begin{aligned}\tilde{s} &= (\tilde{x}_1+\tilde{x}_2)(1+\epsilon_3)\\ &= x_1(1+\epsilon_1)(1+\epsilon_3)+x_2(1+\epsilon_2)(1+\epsilon_3)\\ &\approx x_1(1+\epsilon_1+\epsilon_3)+x_2(1+\epsilon_2+\epsilon_3)\\ &\approx s(1+2\epsilon)\end{aligned} \tag{3.33}$$

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: $1.23\cdot10^{0}\times5.67\cdot10^{1}=0.69741\cdot10^{1}\rightarrow 6.9741\cdot10^{0}\rightarrow 6.97\cdot10^{0}$.

Exercise 3.17. 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:

$$\begin{array}{rll} .5 \times 2.47 - 1.23 & = 1.235 - 1.23 & = 0.005 \quad \text{exact} \\ \downarrow & & = 5.0 \cdot 10^{-3} \\ 1.24 - 1.23 & = 0.01 = 1. \cdot 10^{-2} \quad \text{compute} & \end{array} \tag{3.34}$$

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 3.18. Consider the iteration

$$x_{n+1} = f(x_n) = \begin{cases} 2x_n & \text{if } 2x_n < 1 \\ 2x_n - 1 & \text{if } 2x_n \ 1 \end{cases} \tag{3.35}$$

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

Another effect of the way floating point numbers are treated is on the *associativity* of operations such as summation. While summation is mathematically associative, this is no longer the case in computer arithmetic.

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

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.

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 3.19. 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, 190].

3.6.1 Cancellation: the 'abc-formula'

As a practical example, 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 $x_+ \cdot x_- = c/a$.

Exercise 3.20. Explore computing the roots of

$$\epsilon x^2 - (1 + \epsilon^2)x + \epsilon \tag{3.38}$$

by the 'textbook' method and as described above.

- What are the roots?
- Why does the textbook method compute one root as zero, for ϵ small enough? Can you think of an example where this is very bad?
- What are the computed function values in both methods? Relative errors?

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

$$\begin{aligned} \sin(x) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \\ &= \textstyle\sum_{i\geq 0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}. \end{aligned} \tag{3.43}$$

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 13 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. This 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, ordinary execution corresponds 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 [122, 123]. 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 3.8.7 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 models*, for the actual sizes of these.

- *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. 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` (!!!), and 64 bits for pointers. This is used in the *Win64* API.
- *LP64* (or '4/8/8') has 32-bit `int`s, and 64 bits for `long` and pointers.

- *ILP64* (or '8/8/8') uses 64 bits for all of `int`, `long`, and pointers. This only existed on some early Unix systems such as *Cray UNICOS*.

Also, see https://en.cppreference.com/w/cpp/language/types.

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:

```
// printbits.c
void printBits(size_t const size, void const * const ptr)
{
  unsigned char *b = (unsigned char*) ptr;
  unsigned char byte;
  int i, j;

  for (i=size-1;i>=0;i--)
    for (j=7;j>=0;j--) {
      byte = (b[i] >> j) & 1;
      printf("%u", byte);
    }
}
```

Sample usage:

```
// bits.c
  int five = 5;
  printf("Five=%d, in bits: ",five);
  printBits(sizeof(five),&five);
  printf("\n");
```

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:

```
// 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 25.4* for details.)

For instance

```
std::numerical_limits<int>.max();
```

3.7.4 Exceptions

Both the IEEE 754 standard and the C++ language define a concept *exception* which differ from each other.

- Floating point exceptions are the occurrence of 'invalid numbers', such as through overflow or divide-by-zero (see section 3.4.1.1). Technically they denote the occurrence of an operation that has 'no outcome suitable for every reasonable application'.
- Programming languages can 'throw an exception', that is, interrupt regular program control flow, if any type of unexpected event occurs.

3.7.4.1 Enabling

Sometimes it is possible to generate a language exception on a floating point exception.

The behavior on *overflow* can be set to generate an *exception*. In C, you specify this with a library call:

```
#include <fenv.h>
int main() {
 ...
 feenableexcept(FE_DIVBYZERO | FE_INVALID | FE_OVERFLOW);
```

3.7.4.2 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 `limits`, 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.

3.7.4.3 *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 3.8.2.
- 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 3.10 above. We use the concept of *value safety* to describe

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.

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.

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 1.7.3. 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 [137].

3.8 More about floating point arithmetic

3.8.1 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.

```
// denormal.cxx
    /* init data */
    for ( int t=0; t<stream_length; t++)
      memory[t] = startvalue;

    /* repeated scale the array */
    for (int r=0; r<repeats; r++) {
      for ( int t=0; t<stream_length; t++) {
        memory[t] = ratio * memory[t];
      }
      memory.front() = memory.back();
    }
```

The left column shows that the *Intel i5* of a MacBook Air does not have hardware support for denormals. The next two columns shows the *Intel Cascade Lake* but with two different compiler settings: First we use the default behavior of *flush-to-zero*: any subnormal number is set to zero; then we show the slowdown associated with correct handling of denormals.

```
This float type:
min normal=1.175494e-38
    eps=1.192093e-07
progression from 1 with ratio: 0.9
  final result: 0.00515377
  nsec per access: 1.176
progression from 1.17549e-38 with ratio: 0.9
  final result: 6.05823e-41 (underflow)
  nsec per access: 47.567
```

```
This float type:
min normal=1.175494e-38
    eps=1.192093e-07
progression from 1 with ratio: 0.9
  final result: 0.00515377
  nsec per access: 0.637
progression from 1.17549e-38 with ratio: 0.9
  final result: 0 (flushed to zero)
  nsec per access: 0.646
```

```
This float type:
min normal=1.175494e-38
    eps=1.192093e-07
progression from 1 with ratio: 0.9
  final result: 0.00515377
  nsec per access: 1.321
progression from 1.17549e-38 with ratio: 0.9
  final result: 6.05823e-41 (underflow)
  nsec per access: 42.926
```

3.8.2 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*[114], 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.

3.8.3 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' [111], 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.

3.8.4 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 80287 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 [161].

3.8.5 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'.

3.8.5.1 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.

3.8.5.2 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 *Deep Learning (DL)* 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 `https://en.wikipedia.org/wiki/Bfloat16_floating-point_format` 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.

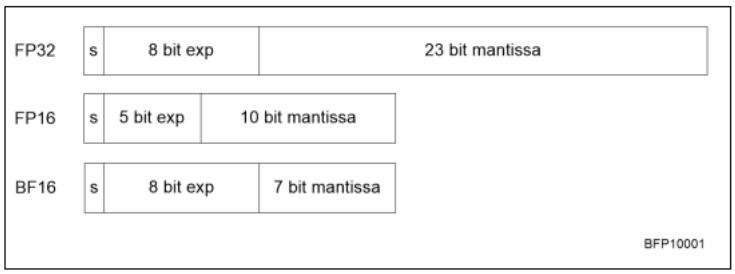


Figure 3.4: Comparison of fp32, fp16, and bfloat16 formats. (Illustration from [36].)

- 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.

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].

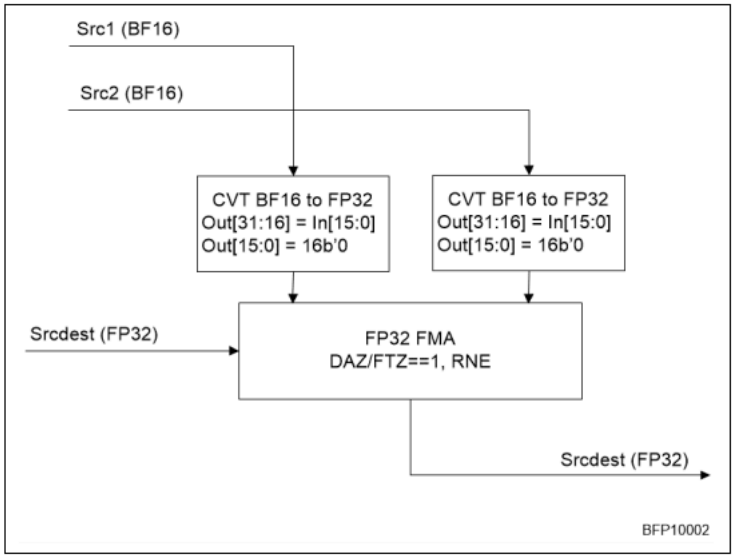


Figure 3.5: An FMA unit taking two bloat16 and one fp32 number. (Illustration from [36].)

3.8.6 Fixed-point arithmetic

A fixed-point number (for a more thorough discussion than found here, see [194]) 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.

3.8.7 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 1.3.4.7). 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.

5. These two header files are not identical, and in fact not compatible. Beware, if you compile C code with a C++ compiler [52].

- Even rearrangements of the same computations do not behave identically; see the summing example.

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.