Introduction to Supercomputing

TMA4280

Einar Rønquist
Arne Morten Kvarving
Eivind Fonn

Chapter 1

Introduction

1.1 Supercomputing

What do we mean by supercomputing? In general, we mean using computers to solve problems which are very computationally intensive, i.e. problems which require large computing resources in terms of memory or floating-point operations, or both. Examples of problems requiring supercomputing are weather forecasting (http://met.no uses the supercomputing facility here at NTNU), oil exploration (reservoir simulation), seismic analysis (an inverse problem), computational chemistry, computational physics, computational mechanics (e.g. structural mechanics, fluid mechanics) and material science.

Due to the resource demanding aspects of supercomputing, it is important to make the whole solution process as efficient as possible. Examples of important issues to study are:

- numerical and computational algorithms which are fast, robust and accurate
- software development
- treatment of large data sets
- visualization
- validation of the simulation results

Supercomputing typically implies the use of parallel processing. This means that the underlying algorithms used need to be designed and tuned for such a computing environment. The software development typically becomes more involved compared to a single-process implementation even though the trend is to abstract the machine specific details as much as possible.

Because of the compute-intensive (memory, floating-point, I/O) nature of these problems, they often require special high-end computing platforms. In Norway, there is at least one such system at each major university. The

large oil companies in Norway also have such computing platforms. Each system is still quite expensive to purchase, to operate and to maintain. In addition, special intrastructure often needs to be built around such systems: special rooms (security), power supply, cooling systems etc. Due to the overall investment, both in terms hardware and in terms of human resources, it is important to use these systems as efficiently as possible—each system typically lasts only a few years.

Even if the current systems are powerful, it is important to make progress in terms of making such systems easier to build and to use, and to develop more efficient computational algorithms. This will enable larger and more realistic problems to be simulated, which translates into progress in science and engineering (which are the "classical" applications for supercomputing), but also in other emerging areas such as the medical field.

In the following, let us briefly mention some of the issues involved when working with a specific application. Assume that we have a program for simulating the blood flow through part of a vein. Let T_1 be the solution time on a single processor (even though, in reality, such a simulation would probably not fit on a single-processor machine).

Assume now that we port this simulation to a multi-processor system with P processors. Ideally, we would like the solution time to be reduced by a factor of P. Another way of saying this is that we would like to achieve a speedup

$$S_p = T_1/T_p = P, \tag{1.1}$$

where T_p is the solution time on P processors. Typically, the speedup $S_p < P$. As we increase the number of processors to solve our particular (fixed) problem, we will first see a good speedup, followed by a degradation as we add more and more processors. The main explanation for this is easy to understand. In order to achieve a perfect speedup, we cannot have any overhead when moving from a single processor system to a multi-processor system. In reality, we have communication between the processors. As the number of processors is increased, the work per processor goes down, while the overall communication overhead goes up. A critical issue is therefore to minimize the communication overhead so that we can take advantage of larger systems for a fixed problem.

We should here remark that one of the major advantages of having access to a supercomputing facility is the possibility to solve much larger problems than what is possible on a smaller system. Hence, as we increase the number of processors, we may also increase the problem size so that the work per processor remains fixed. We will consider some of these issues in this course.

Let us now discuss the single-processor performance a bit more. Assume for the moment that we have two different versions (executables), A and B, of a particular application. We also assume that the underlying numerical algorithms are identical. The simulation times for the two versions are denoted

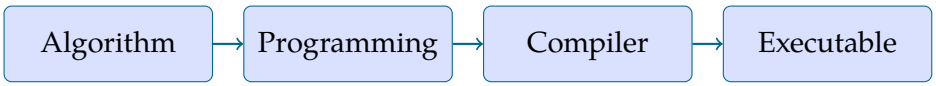


Figure 1.1: Ingredients in a simulation.

as T_1^A and T_1^B, respectively. We observe that

$$T_1^A = 3T_1^B. \tag{1.2}$$

Why are the two simulation times so different since the numerical algorithms are the same?

We only mention a few aspects which may have caused this; see Figure 1.1:

- different choice of data structure / memory layout during the programming;
- different use of optimized numerical libraries;
- different choices of compiler optimization.

We will look at some of these issues in this course. Note that it is typically harder to achieve a good speedup for version B compared to version A, even though $T_1^B < T_1^A$ (can you suggest a reason why?)

Are there other ways to change the solution time for our particular simulation? Yes; we mention two distinct ways:

- change the computer;
- change the computational or numerical algorithms.

Progress in the available hardware has been impressive over the past decades. However, it is important to realize that progress in computational algorithms has been equally impressive. Note that developing a new algorithm which requires only half the number of floating point operations is equally important as buying a new computer with twice the performance in terms of floating point operations per second (Flops).

We will discuss a few selected numerical algorithms in this course, e.g. numerical solution of partial differential equations (primarily finite difference methods), iterative and direct methods for solving linear systems of equations, basic linear algebra routines, and a little bit about statistical methods (Monte Carlo methods).

Figure 1.2: Each floating point has a binary representation with three fields: S denotes the sign of the number, E is an exponent, and F is the fraction part of the mantissa.

Table 1.3: Number of bits for common floating point standards.

Precision	S	E	F	Total
Single	1	8	23	32
Double	1	11	52	64

1.2 Floating point representation

This section gives a brief introduction to the IEEE standard (IEEE-754) for floating point representation. A basic understanding of the standard is appropriate given the importance of floating point operations in this course.

All real numbers are represented with a finite number of bits. The representation is depicted in Figure 1.2. The sign bit is certainly easy to understand. A value $S = 0$ means that the number is positive, while a value $S = 1$ means that the number is negative.

The actual decimal value V of the floating point is

$$V = (-1)^S \cdot 2^{E-B} \cdot M \tag{1.3}$$

where M is the mantissa and B is denoted as the *bias* which will be explained below. The exponent E is always adjusted such that the mantissa can be expressed as

$$M = \underbrace{1}_{\times 2^0} . \overbrace{\underbrace{b_1}_{\times 2^{-1}} \underbrace{b_2}_{\times 2^{-2}} \ldots}^{F} \tag{1.4}$$

where F denotes the fraction of the mantissa in binary representation (i.e. the binary digits $b_1 b_2 \ldots$). Since this representation is always assumed, the binary number 1 in the front is not explicitly represented. Note that,

$$1 \leq M < 2, \tag{1.5}$$

since the fraction F is always less then one.

There are many more details regarding floating point representation. For example, the implicitly assumed binary number 1 in the mantissa is only true for what is referred to as *normalized* numbers.

A few words about the exponent E. In single precision, E is represented using 8 bits, giving 256 possibilities, of which 254 are used to represent

normalized numbers. The values $E = 0$ and $E = 255$ are primarily reserved for zero and infinities. The actual exponent used to find the corresponding decimal value of a normalized number is $E - B$; see equation (1). For single precision, $B = 127$ and for double precision, $B = 1023$.

We can now compute the maximum and minimum numbers that we can represent in single precision (i.e. using 32 bits):

$$V_{\max} = 1 \cdot 2^{254-127} \cdot 2 \approx 3.40 \cdot 10^{38} \tag{1.6}$$

$$V_{\min} = 1 \cdot 2^{1-127} \cdot 1 \quad \approx 1.17 \cdot 10^{-38} \tag{1.7}$$

As we can see, this is a significant range.

Having considered the range, let us now consider the accuracy of a given floating point number. Again, consider the single precision case. The fraction of the mantissa is represented using 23 bits, meaning that the smallest *fraction* that can be represented is

$$2^{-23} \approx 1.19 \cdot 10^{-7}.$$

This implies that *any* floating point number using single precision (covering the whole range discussed above) has about 7 digits of accuracy.

Exercise 1.1 *Consider the maximum and minimum numbers defined in* (1.6)-(1.7). *How many digits should we include in each of these numbers when written out?*

Exercise 1.2 *Find the binary floating point representation of the decimal number* 4.25 *in single precision.*

Exercise 1.3 *How many decimal digits of accuracy does a double precision floating point number have?*

1.3 Integer representation

An integer is typically represented using 32 bits. One bit is used to represent the sign. Hence, the possible integers will be in the range $\pm 2^{31} \approx \pm 2 \cdot 10^9$.

If an algorithm includes a loop which needs to be done n times (e.g., a Monte Carlo simulation), n needs to be less than approximately 10^9. This may seem sufficient, however, there could be situations when this limitation could become an issue.

Exercise 1.4 *Propose one or several ways around this limitation.*

1.4 Floating point performance

The basic operations of adding, multiplying, subtracting and dividing two floating point numbers are counted as floating point operations.

Floating point performance is usually measured in number of floating point operations per second, commonly abbreviated as Flops; for a brief discussion see http://en.wikipedia.org/wiki/FLOPS.

Since current computer systems typically perform many of these operations per seconds, abbreviations like MFlops (megaflops), GFlops (gigaflops), TFlops (teraflops), and PFlops (petaflops) are common to use.

When we speak about the speed of a computer, we will typically mean the floating point performance.

Exercise 1.5 *Let c be a scalar (a floating point number), let $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ be vectors, each comprising n floating point numbers, and let $\boldsymbol{A}$ be an $n \times n$ matrix. How many floating point operations does it take to perform the following basic linear algebra operations?*

$$\boldsymbol{z} = \boldsymbol{x} + c\boldsymbol{y}, \qquad \boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$$

1.5 Storage requirements

It is quite common today to do all the necessary computations using double precision. This means that 64 bits are used to represent each floating point number, which is equivalent to 8 bytes (8 bits per byte).

Let us now consider how many floating point numbers we can store in the memory associated with a typical processor. Assume that we have available 4 GB ($4 \cdot 10^9$ bytes) of main memory (or RAM: *Random Access Memory*); this is a quite typical size nowadays. This amount of memory is sufficient to represent a total of 512 millon floating point numbers in double precision.

In practice, we could never use the entire memory for such a purpose. We also need space for our simulation program (instructions etc.). However, it gives an indication of the order of magnitude that we can handle.

If we are solving differential equations or some other problem, we also need to use memory to store matrices, temporary variables, etc. The available memory to store the actual solution depends on the numerical algorithm, however even a fairly scalable algorithm would typically only be able to handle a few million unknowns. Many algorithms would allow far fewer unknowns.

Problems in science and engineering can easily require many million unknowns, perhaps even billions of unknowns. In order to solve such problems, we obviously need both more memory and increased computing power. This is achieved using parallel processing where multiple processors and memory modules are connected into a single computational platform.

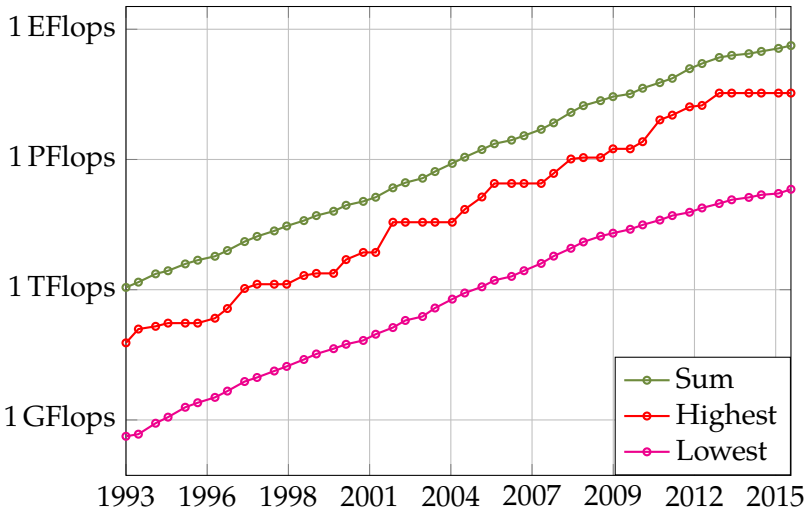


Figure 1.4: The performance development since 1993 for the top 500 supercomputers in the world. In particular, the figure indicates the performance development for the fastest system (number 1), the slowest system (number 500), and the sum of all the 500 fastest systems.

Exercise 1.6 *Let $\boldsymbol{A}$ be an $n \times n$ matrix, and $\boldsymbol{x}$ and $\boldsymbol{b}$ be two vectors of length n. Assume that we want to solve the linear system of equations*

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

using Gaussian elimination. Assume further that the matrix $\boldsymbol{A}$ is dense, meaning that we need to store all the n^2 entries in the matrix.

What is (approximately) the largest equation system we can solve (i.e., the largest number of n we can use) and still be able to fit the whole problem in the main memory, which we assume is 1 GB*?*

1.6 Past, current and future supercomputers

Figure 1.4 depicts the past and future performance development of the 500 fastest supercomputers in the world. More information about the top 500 systems can be found on http://top500.org.

In 1986 a supercomputing center was established at NTH. Table 1.5 gives a summary of the supercomputers at NTH/NTNU since then.

The current supercomputer at NTNU is an SGI Altix system with 20736 (physical) processors. For an example of what such a system looks like, see Figures 1.6 and 1.7. Some of the exercises in this course will be done using this system. Some of the key characteristics of this system are:

- Full name: `vilje.hpc.ntnu.no`

Table 1.5: Supercomputers at NTH/NTNU. Note that some of this information is guesswork as not all machines were not available at when revising this document.

Year	System	Processors	Type	GFlops
1986–1992	Cray X-MP	2	Vector	0.5
1992–1996	Cray Y-MP	4	Vector	1.3
1995–2003	Cray J90	8	Vector	1.6
1992–1999	Intel Paragon	56	MPP	5.0
1996–2003	Cray T3E	96	MPP	58
2000–2001	SGI O2	160	ccNUMA	100
2001–2008	SGI O3	898	ccNUMA	1000
2006–2011	IBM P5+	2976	Distributed SMP	23500
2012–	SGI Altix ICE X	23040	Distributed SMP	497230

- System: SGI Altix ICE
- Type: Distributed SMP
- Number of nodes: 1440
- Nodes: Shared memory system; 2 octa-core chips; 32 GB memory
- Number of cores (processors): 23040
- CPU type: Intel Sandy Bridge
- Theoretical peak performance: 479.23 TFlops
- Weight: This machine is shy and refuses to tell me

1.7 Applications

Some sample applications for supercomputing are given in Figures 1.8, 1.9 and 1.10.

Figure 1.6: *Njord*, the previous IBM supercomputer system at NTNU. Courtesy of Arve Dispen.

Figure 1.7: Photo of *Vilje*, the current SGI supercomputer at NTNU. Source: NTNU HPC Wiki.

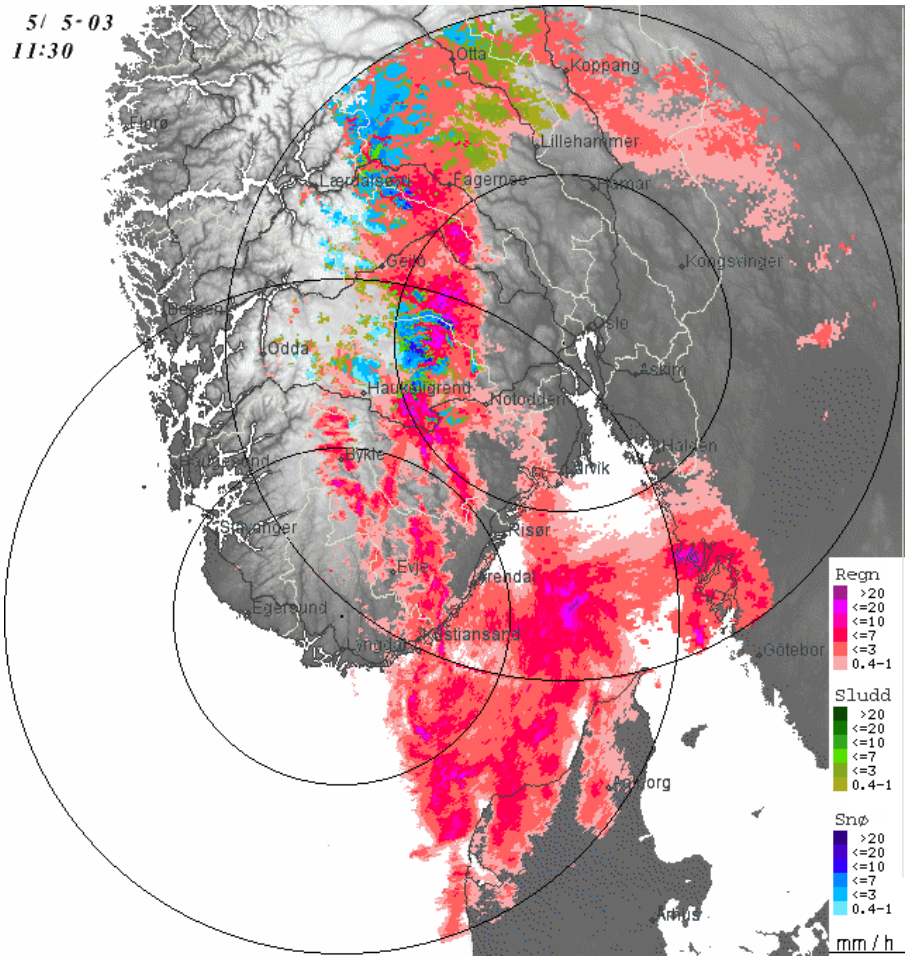


Figure 1.8: Example of use of supercomputing: weather forecasting. The Meteorological Institute in Norway uses the supercomputer *Njord* every day. Source: http://met.no.

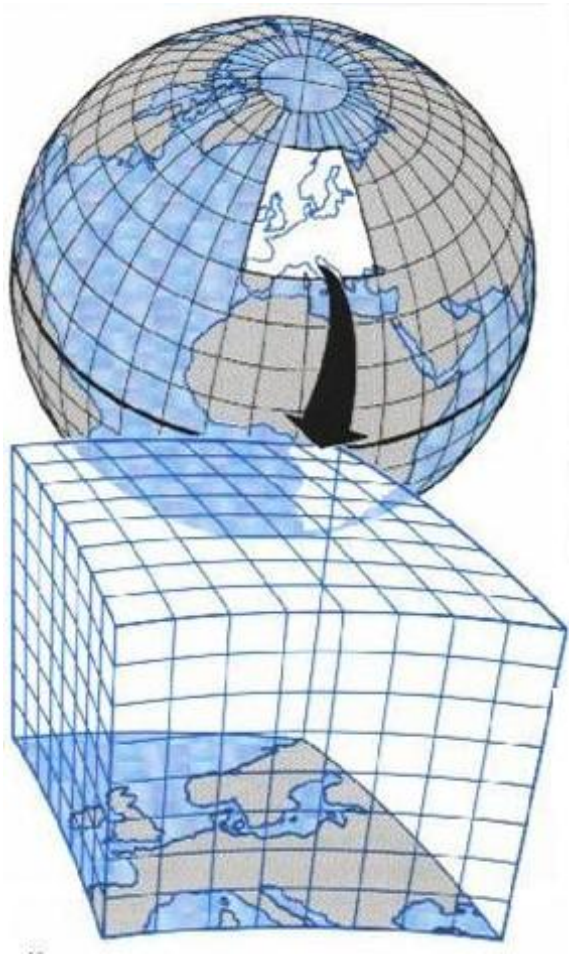


Figure 1.9: Example of use of supercomputing: climate modelling. For more information, see http://bjerknes.uib.no (Bjerknes Centre for Climate Research).

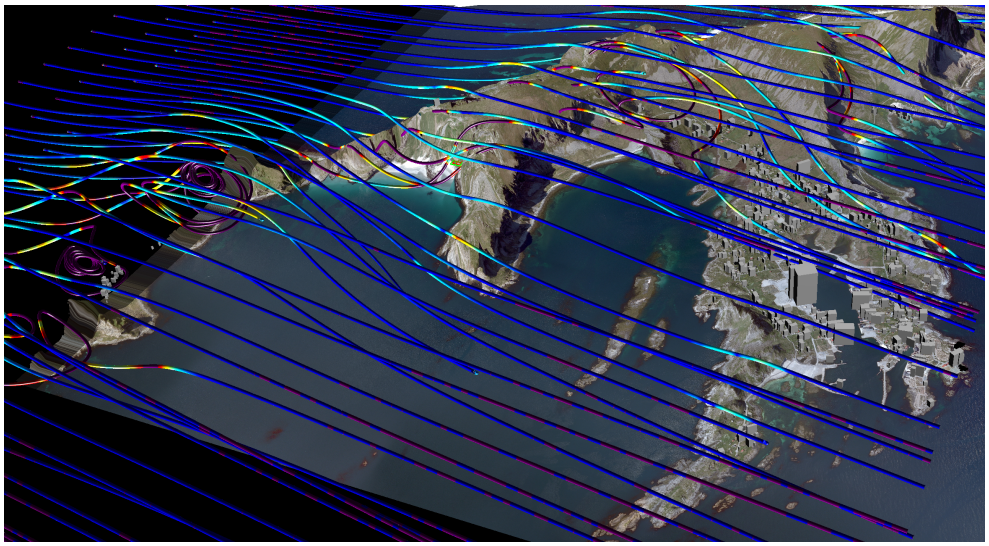


Figure 1.10: Example of use of supercomputing: turbulence modelling.

Chapter 2

Single processor systems

2.1 A prototypical processor

We start by explaining some of the basic tasks performed by a single processor. The comments are particularly relevant for the MIPS R14000 processor used in a previous supercomputer at NTNU (called *Gridur*, a SGI Origin 3000 system used in the period 2001–2007). The MIPS processor is an example of a processor implementing a RISC architecture (RISC—*Reduced Instruction Set Computer*), which has been a very important processor design over the past couple of decades. We will later return to comment on the POWER5 processor used in the previous supercomputer at NTNU, in particular some of the key differences compared to the processor discussed in this section. The name POWER refers to *Performance Optimization With Enhanced RISC*, and is also the name of a series of microprocessors designed by IBM.

In order to introduce some key concepts in the RISC architecture, let us briefly explain what happens when we perform the following simple operation

$$c = a + b. \tag{2.1}$$

Here, we want the processor to add the two numbers a and b and store the answer in c. In this case, a and b are referred to as *operands*, while + is the *operator*. Hence, the simple addition of two scalars implies a single floating point operation.

The basic unit of "time" for our processor is a clock cycle. The state of the processor changes from clock cycle to clock cycle, depending on what needs to be done. Different parts/units of the processor work on different tasks independently of each other. These tasks may correspond to different instructions, each in a particular phase of completion.

Consider again the addition of two scalars. The operation (2.1) may be part of a larger program involving many operations (or instructions). Let us assume that the instruction for the "add" operation is currently in a small

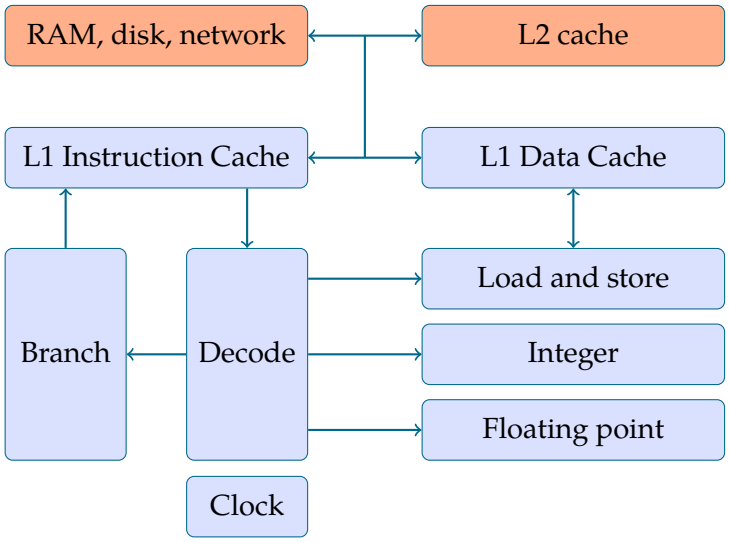


Figure 2.1: A prototypical processor, including the MIPS R14000 used in *Gridur*, the supercomputer at NTNU during the period 2001–2007. The red components are off-chip.

memory module denoted as L1 cache (for instructions). When the execution of our program is ready to perform this operation, the instruction is brought into a decoder and made ready for execution. The addresses of the operands (a and b) are computed and the operands are brought into two registers of the processors. We assume that the operands are available in the small memory module called L1 cache (for data).

The operands are then directed from the two registers to the floating point unit for addition (denoted as FPAdd) where the two numbers are added together. The whole operation takes just a few clock cycles. For the MIPS R14000 processor, this operation takes five clock cycles:

1. read from register
2. align
3. add
4. pack
5. write to register

Note that the number of clock cycles required to perform this operation is not the same as the number of bits used to store our operands (e.g., 64 bits in double precision). The reason is that the data flow in the processor happens along a "wide bus" capable of moving all the bits at the same time. This is an example of bit-level parallelism. Older processors could only move 4, 8,

16, and 32 bits at a time and an add operation therefore took additional clock cycles to complete. In addition, current processors have a much higher clock frequency (or shorter clock period) compared to older processors.

Let us now make a few more remarks regarding the processor in Figure 2.1. We have already mentioned the floating point unit for addition, FPAdd. However, the processor has also other functional units. For example, the R14000 prosessor has five functional units which can operate independently from each other:

- Load/Store: computes memory addresses and to bring operands to and from the memory;
- ALU1: a unit for addition and subtraction of integers, and logical operations;
- ALU2: a unit for addition, subtraction, multiplication, and division of integers, as well as logical operations;
- FPAdd: adds of floating point numbers;
- FPMult: a unit for multiplication, division, and square root of floating point numbers.

Note that the operands used in these functional units need to come from the registers in the processor. If the operands are not in the registers, they have to be brought from the memory into the registers with a *load* operation. The answer (or output) from the functional units are also stored in registers and subsequently stored in the memory with a *store* operation.

2.2 Memory hierarchy

The example from the previous section involving the addition of two floating point numbers assumed that the instruction for the operation (2.1) was in the memory module denoted as L1 cache for instructions; see Figure 2.1. Similarly, we assumed that the operands a and b were available in the memory module denoted as L1 cache for data.

Let us now comment a bit more on what happens if these assumptions are not true. In this case, the program will check whether the instruction or data are available in the memory module denoted as L2 cache in Figure 2.1. This is a memory module which is larger than L1 cache. Furthermore, the L2 cache is not split into a separate module for instructions and a separate module for data. It also takes longer (meaning more clock cycles) to fetch data from L2 cache compared to L1 cache. It could also happen that the instruction and the data the program is looking for is not available in L2 cache either. In this case data has to be brought in from main memory (RAM) or perhaps even further away (e.g. the local disk).

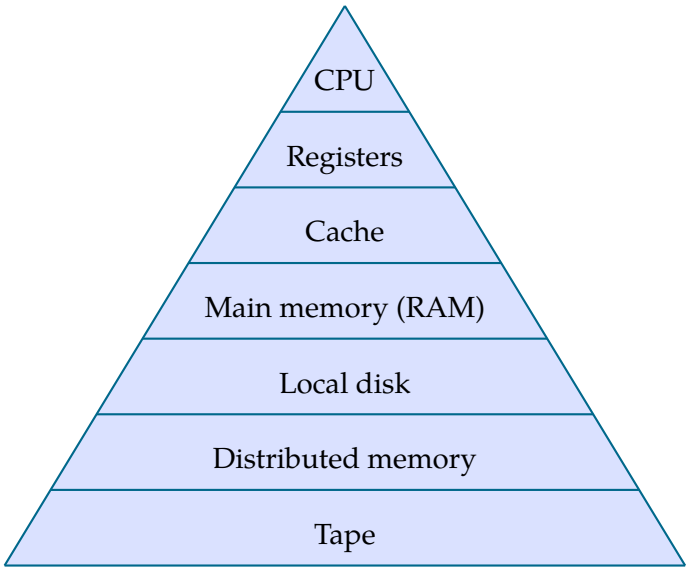


Figure 2.2: The memory hierarchy of a computer system. Higher entries are faster, while lower entries are cheaper and larger.

Bringing instructions and data to and from memory (from cache, RAM, etc.) is typically a bottleneck in scientific computing. The processors are getting faster and faster, but the memory bandwidth (or transfer rate in bytes per second) is not keeping up at a similar speed. Hence, for certain operations, the processor can become "starved" for data, meaning that it is idle for much of the time while waiting for data to be transferred to and from memory. The overall performance in terms of floating point operations per second will in such cases not be governed by the processor speed, but by the memory access time.

In order to "hide" this difference in speed (i.e., processor speed versus memory access speed), the memory is organized in a hierarchical fashion; see Figure 2.2. The fastest part of the memory is closest to the CPU. For example, on the MIPS R14000, the L1 cache is on the chip. This part of the memory is fast, but also small because it is very expensive. In contrast, the main memory is much larger, but has also a much longer access time; see Table 2.3. We will later discuss in more detail how it is decided what should be in the L1 and L2 cache.

We remark that message passing in Table 2.3 refers to communication between individual processors by sending messages over a network; we will return to a discussion of multiple processors later.

Table 2.3: Typical memory access times for R14000. The numbers represent number of clock cycles.

Memory type	Clock cycles
Registers	1
L1 cache	2–3
L2 cache	10–12
Main memory	100–200
Message passing	$O(10^3)$–$O(10^4)$
Local disk	$O(10^6)$

2.3 Pipelining/vectorization

Assume now that, instead of (2.1), we would like to add the two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ of length n and store the result in a vector $\boldsymbol{c}$, i.e.

$$\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}. \tag{2.2}$$

We can also write (2.2) as the loop (in Fortran)

```
for i=1,n
  c(i) = a(i) + b(i)
end
```

Again, we start by assuming that the data are available in the registers. From there, the individual vector elements $a(i)$ and $b(i)$ enter the floating point unit FPAdd where the numbers are added to produce the output elements $c(i)$, $i = 1, \ldots, n$.

We mentioned earlier that it takes five clock cycles to add two floating point numbers together. This would perhaps suggest that the total number of clock cycles for the operation (2.2) is $5n$, and that the total execution time therefore is $5n\tau$ where τ is the clock period. However, if things are done optimally, the total number of clock cycles can be reduced to approximately n for $n \gg 1$. The reason for this is that the five stages in the adder correspond to independent tasks. This means that, as soon as the addition of two operands (i.e. $a(i)$ and $b(i)$) has finished the first stage, two new operands ($a(i+1)$ and $b(i+1)$) can enter the first stage in the adder. Hence, after five clock cycles, the first number $c(1)$ in the operation (2.2) is ready, while the numbers $c(2)$, $c(3)$, $c(4)$, $c(5)$, and $c(6)$ are in different phases of completion in the adder.

In summary, after a few clock cycles to "fill up" the adder, a new answer $c(i), i = 2, \ldots, n$ is ready every clock cycle. This is what is referred to as *pipelining* (or sometimes *vectorization*). The reason behind this term is quite obvious: we constantly feed the floating point unit (in this case, the adder) so

that the "pipeline" is always full. In other words, we do not wait until one answer is ready before we start the process of adding two new numbers. In this way, we achieve a certain level of parallelism in the sense that asymptotically (for long vector lengths), the adder works simultaneously on five different pairs of operands.

The above discussion assumed that the data (i.e. the operands $a(i)$ and $b(i)$, $i = 1, \ldots, n$) are ready for the adder with no delay. Whether this is possible or not depends on the particular processor. In the case of the MIPS R14000 processor it is not possible to achieve this performance. The reason is that, in the best case, only a single floating point number can be brought between the memory (L1 cache) and a register at a time. Since we need to fetch two operands, $a(i)$ and $b(i)$, per floating point operation, and store the answer $c(i)$ back to memory, a minimum of three clock cycles are needed for memory transfer per addition. Hence, even though the floating point unit (the adder) can theoretically complete one addition per clock cycle, the memory traffic will be the bottleneck, at least for large n.

The only possibility for achieving a better performance is if all the operands are already available in the processors registers. However, since a processor only has a limited number of registers (on the MIPS R14000 the number of registers is 64), such performance cannot be acheived if $n \gg 1$.

Let us now predict the optimal performance for the operation (2.2) in the case $n \gg 1$ on Gridur. The clock cycle for each processor is 500 MHz. Hence, the clock period is 2 ns. Since each floating point operation will asymptotically require three clock cycles due to the fetch and store operations (see the discussion above), each floating point operation will at least require three clock cycles, or 6 ns. Hence, the maximum performance for the operation (2.2) is $(6 \cdot 10^{-9})^{-1}$ floating point operations per second, or approximately 167 MFlops. In practice, less performance may be achieved, in particular if the operands need to first be brought in from deeper layers of the memory hierarchy; see Figure 2.2.

2.4 Superscalar operations

Consider now a modification of the operation (2.2) to the following operation:

$$\boldsymbol{c} = \boldsymbol{a} + \gamma \boldsymbol{b}. \tag{2.3}$$

Here, each vector element $b(i)$, $i = 1, \ldots, n$ is multiplied with a scalar, γ, before being added to $a(i)$. Similar to the operation (2.2) the result of each addition is stored as the vector element $c(i)$.

The new operation here is the multiplication. As mentioned earlier, each R14000 processor has a separate floating point unit for multiplication, FPMult. Similar to the add operation, multiplying two numbers also take five clock cycles:

1. read from register
2. multiply
3. sum product
4. pack
5. write to register

Hence, all the comments made in the previous section for the operation (2.2) also apply if the add operation + is replaced by multiplication ×.

Let us now comment on what happens when we combine both multiplication and addition as in (2.3). Again, let us first assume that all the data are readily available (i.e. stored in the registers). The vector elements $b(i)$ are brought to the multiplier where each element is multiplied by the scalar γ; see Figure Figure 2.4. After a startup time of five clock cycles, a new answer is coming out from the multiplier every clock cycle. We assume here that the pipelining feature is exploited.

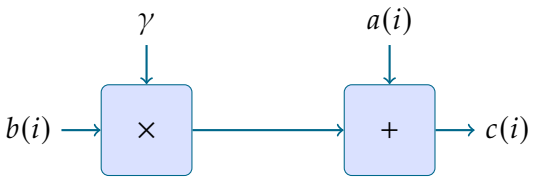


Figure 2.4: The superscalar operation multiply and add.

Each output from the multiplier is now channeled directly as an input to the adder where it is added to the vector element $a(i)$. After another five clock cycles, the answer $c(i)$ is ready. Hence, after a startup time of 10 clock cycles (5 for the multiplier and 5 for the adder), we get one complete answer $c(i)$, $i = 1, \ldots, n$ as output every clock cycle. Asymptotically (i.e., for large vector lengths), the theoretical performance is therefore two floating point operations per clock cycle. This way of piping the output from one floating point unit into the input for another unit is denoted as *superscalar* capability. Similar to the pipelining feature of the adder and the multiplier, the superscalar capability offers yet another possibility of parallelism in the sense that each single processor is capable of performing addition and multiplication at the same time (for sufficiently long vector lengths).

In practice, the processor has only a limited number of registers, and we need to fetch the operands from memory (L1 cache) and store the answers in memory. Similar to the operation (2.2), each complete vector element $c(i)$ will require three clock cycles due to the memory traffic. However, in contrast to the operation (2.2), the operation (2.3) implies two floating point operations

instead of one for each complete vector element $c(i)$. The maximum single-process performance we can obtain on R14000 for (2.3) is thus twice the performance for (2.2), i.e. 333 MFLOPS.

2.5 Cache

Let us now discuss in more detail the interaction between the cache and the main memory. The main purpose of the cache is to keep copies of data in extra (and fast) memory close to the CPU in order to "hide" the relatively slow transfer rate between the main memory and the processor.

Because fast caches are expensive, they tend to be small. As an example, we give the memory sizes for the R14000 processors. On Gridur, four individual processors shared up to 4 Gbytes of main memory. Each processor had an L2 cache of size 8 Mbytes and two L1 caches (one for instructions and one for data), each only of size 32 Kbytes. Hence, the L1 cache for data can only hold up to 4000 floating point numbers (assuming double precision), which is relatively small in the context of simulating systems with thousands or millions of unknowns (e.g., for the numerical solution of partial differential equations). The L2 cache can hold more data, but the transfer rate is a little bit longer compared to the L1 cache; see Table 2.3.

The cache is smaller than the main memory by some power of two. Hence, a strategy for mapping memory locations to cache locations needs to be defined. We describe three strategies for doing this.

Direct mapped cache

One strategy is to use what is refered to as a *direct mapped cache*. In this case, each location in main memory corresponds to a unique location in cache; see Figure 2.5. The main memory address is split into two parts: the first bits of the memory address are called the *set bits* and these bits give the precise cache address. The remaining bits are called the *tag bigs*, and these are used to determine if a copy of the content at the particular main memory location has been copied into the cache location given by the set bits. With this strategy we see that several main memory addresses map to the same cache address.

$$\text{Memory address} = \underbrace{b_1 \ldots b_k}_{\text{tag bits}} \underbrace{b_{k+1} \ldots b_N}_{\text{cache address}} .$$

When some particular data is requested by the program, e.g., a floating point number, the processor will check whether the data is stored in L1 cache. It does this by looking up the cache address (taking the least significant bits of the memory address) and checking whether the tag bits at that location match the tag bits of the memory address. If is not in L1 cache, the processor will check whether the data is stored in L2 cache. If this is the case, the requested

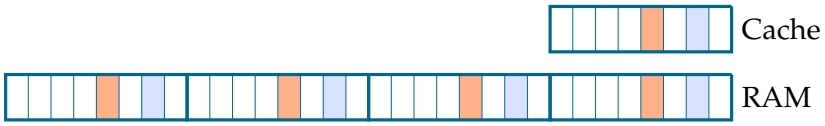


Figure 2.5: A direct mapped cache. Each main memory address maps to a unique and pre-determined location in the cache.

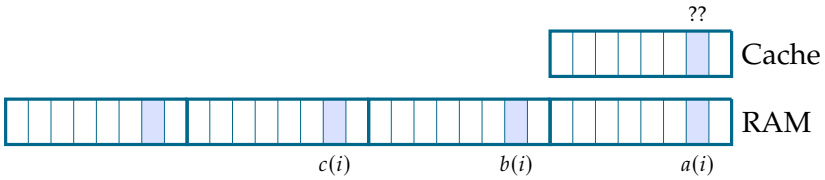


Figure 2.6: Cache trashing. The corresponding elements in the vectors $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ all map to the same cache address.

data will be copied from the L2 cache into L1 cache. If the data is not in L2 cache either, the data will have to be brought in from main memory. In this case, a copy will be made in L2 cache as well as in L1 cache. In either case, the tag bits at the given cache address will be updated to match the new mapped location.

Note that when a floating point number (or an integer) is requested by the program, more than a single number is copied into cache. The minimum amount of data copied is called a *cache line*. For the R14000 processor, the cache line for the L1 cache is 32 bytes (corresponding to four floating point numbers in double precision), while the cache line for the L2 cache is 128 bytes (corresponding to 16 floating point numbers in double precision). The extra numbers copied are the numbers in the adjacent memory locations in main memory.

Consider again the operation (2.2). Assume that the vectors $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ represent floating point numbers in double precision, and that the vectors are stored after each other in main memory. Let the vector length $n = 4000$, i.e. each vector will precisely fill the L1 data cache. For every element $c(i)$ computed, the operands $a(i)$ and $b(i)$ will need to be brought in all the way from main memory due to the fact that $a(i)$, $b(i)$ and $c(i)$ happen to have the same cache address. A severe drop in performance will be observed in this case. This situation is refered to as cache trashing; see Figure 2.6.

Cache trashing can be avoided by storing the elements in the vectors $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ in a different way; see Figure 2.7. We remark that the crash trashing example given here is perhaps not likely to happen. Nonetheless, it illustrates the point that severe performance degradation is possible to observe due to undesirable memory traffic.

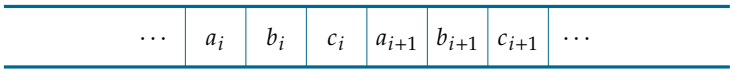


Figure 2.7: Adjacent memory layout.

Fully associative cache

To avoid the possibility of cache trashing, we can use a different cache strategy called a *fully associative cache*. In a fully associative cache, each cache address can map to any memory address. This is essentially a direct mapped cache where all the bits are used as tag bits and no bits are used as a cache address.

To find the corresponding cache location for a given memory address, the tag bits of all cache locations must be checked. This is typically done in parallel using dedicated hardware, which is rather complicated. For this reason, fully associative caches are rarely seen.

When the cache is full, a cache line must be evicted to make way for new data. How this happens depends on the replacement policy:

- Least recently used (LRU);
- Least frequently used (LFU);
- Random

In the context of numerical solution of partial differential equations, the alternative *LRU* generally gives the best performance. This can be understood by the fact that such problems typically exhibit significant locality in time and space: data that has recently been used has a high chance of being used again in the near future; and data close (in space) to recently used data has a high chance of being used in the near future.

Set-associative cache

A *set-associative* cache is a nice compromise between a direct mapped cache and a fully associative cache. In a set-associative cache, the cache is split into chunks of n cache lines. Each memory address maps deterministically to a given chunk (according to its least significant bits) precisely as a direct mapped cache. However, within this chunk the mapping proceeds as with a fully associative cache with n choices.

The cache eviction strategies work the same way as for a fully associative cache.

Chapter 3

Introduction to git

The scope of this chapter is to explain the basic mechanisms of git. Git is a complex tool, using it to its full power can take quite some time to learn. Something crucial may have been missed while attempting to boil it down to basics. There is a whole internet full of guides out there and you are encouraged to supplement these ramblings with more verbose tutorials and articles. Also, please do not hesitate to ask during lectures or breaks; I love it when people talk git to me.

3.1 Installing git

Start by installing git on your laptop; on Ubuntu you can do this through

```
sudo apt-get install git
```

3.2 Setting up a GIT repository

You can set up a GIT directory in two ways. You either clone a remote repository or initialize a new, empty one. In either case you end up on a *branch* called *master* by default.

To clone a remote repository do

```
git clone <url>
```

where <url> is the URL of the repository to clone. E.g.

```
git clone /some/path/on/my/computer
git clone https://github.com/TheBB/TMA4280
```

To initialize an empty repository, change to the directory you would like to use and do

```
git init .
```

3.3 Layout of a git repository

A repository has two parts; the repository information and your local working tree. The first contains information about all the individual commits, commit messages, and branches (a sequence of commits) that is recorded. This is stored in a folder called `.git` in the root of the repository. The second part is your local copy of the files in the repository.

You can make a *bare* clone of a repository. This is a directory that only contains the repository information, i.e. only the *.git* folder of a normal clone, and no local working tree. You do this through

```
git clone --bare <url>
git init --bare .
```

As far as git is concerned, adding files already existing locally is handled as a change to the file where everything was changed (see further down). You just want to add source code, not files generated by the build system or the compiler such as object files, libraries, executables or scripts.

3.4 Keeping track of changes

You can see what changes you have made to your local working directory through

```
git diff
```

If you just want to see which files, are changed, but not the changes themselves, you can use

```
git status
```

This is in general quite useful, it can show more things such as which changes are marked for committing.

To mark some changes for committing, do

```
git add <file>
```

This *stages* the changes, but they have not yet been committed. To commit all staged changes, use

```
git commit
```

This will open a text editor where you can enter a commit message. You can control *which* editor by adding a line such as

```
export EDITOR=<myeditor>
```

to your `~/.bashrc` file if you are using bash. Alternatively, you can specify the commit message directly.

```
git commit -m "my message"
```

You can quickly commit all changes (even unstaged ones) to all tracked files by doing

```
git commit -a
```

The two can be combined:

```
git commit -am "my message"
```

3.5 Keeping track of commits

You can see the commit log through

```
git log
```

You can see the log of commits through

```
git show <commit>
```

The commit hash can be found using git log. Alternatively, you have a few shortcuts. If you omit the revision, the last commit in the branch will be shown. If you want to show the previous commit, you can use

```
git show HEAD~1
```

3.6 Working with remote repositories

A remote repository is a clone of this repository. Since GIT is a distributed revision control system, you can clone a clone, and it still contains all the information the original clone had.

To add a new remote repository, use

```
git remote add <name> <url>
```

where <name> is a name you want to give the remote. If you initialize a repository through cloning another, the repository you cloned will be registered as a remote named *origin*. If you started from scratch, there will be no remotes by default.

To send data between remotes you can *push* or *pull*.

To push changes to a remote, use

```
git push <remote> <branch>
```

If you omit the branch name, all branches will be pushed. If changes you have done conflict with changes done in the remote GIT, your push will be denied. You then have to *pull* from the remote before you push. A push will also be denied if you push to a remote with a local working tree, and you try to push to the branch currently checked out on the remote.

To pull changes from a remote, use

```
git pull --rebase <remote> <branch>
```

If you omit the branch name, all branches will be pulled. Please do not forget the rebase, or you can get yourself in trouble. The pull model is more involved to explain, and deemed outside the scope of this document.

If there are conflicts, git will try to resolve them. If it cannot, you must do it. To see which files are in conflict you can use *git status*. Where there were conflicts you will have something that looks like the following:

```
<<<<<<< HEAD
line11
line22
=======
line12
line21
>>>>>>> 7f8de8cd5f58cd2fa0b2d18f002ce9d9431c0b8c
```

The first line is a starting marker. This is followed by lines that are in conflict, these are from the remote repository. After the equal signs follow the lines in conflict from the local repository. Finally an ending marker and the commit hash from the remote. Change it into what you want it to be and save the file. Remember that there may be multiple blocks in conflict in each file!

Stage the resolved files and continue the rebase by doing

```
git add <file>
git rebase --continue
```

Chapter 4

Multiprocessor systems

4.1 Supercomputing

Supercomputing represents the high performance segment of the overall computing market. This segment has traditionally been driven by grand challenge problems in science and engineering. This is still the situation, although new areas and new applications are constantly being added to the list of problems where supercomputing is necessary (or at least highly desirable).

A strong motivation for using supercomputing is the possibility of performing larger and more realistic simulations, e.g., by solving more detailed mathematical models in science and engineering. Hence, the design of the largest computing systems have traditionally be driven by challenges in science and engineering.

About 30–40 years ago, supercomputers were typically specially designed vector processors capable of performing operations on vector data. A single instruction could operate on multiple data elements (vectors) at once, and the hardware comprised custom-made chips. Because of the low production volume and the costly development effort, each supercomputer was very expensive.

Supercomputers in the 80s and 90s also used fast, expensive, custom-made chips, and combined a few of these in a multiprocess context. However, starting in the late 80s, microprocessor-based supercomputers became more and more popular. These systems were also called *massively parallel processors* (MPPs) because they used many more processors than traditional vector supercomputers. The advantage of the microprocessor-based supercomputers was the use of standard, off-the-shelf microprocessors instead of using costly, custom-made chips. The supercomputer market today is dominated by MPP systems. A supercomputer system typically combines thousands of processors. Some of the largest systems comprise hundreds of thousands of processors.

Supercomputing is very resource demanding in terms of floating point

operations, memory and storage requirement, as well as visualization capabilities. Thus, some of the important challenges related to the development and use of a multiprocessor system concern:

1. communication between the processors
   (network topology, memory access, programming models);

2. development of suitable computational methods or algorithms
   (e.g., domain decomposition algorithms);

3. scalability (both in terms of hardware and algorithms);

4. handling of large volumes of data (storage and visualization).

We add some additional remarks regarding scalability. Let T_P be the time to solve a given problem on P processors, where time refers to wall clock time. We define the speedup, S_P, as

$$S_P = \frac{T_1}{T_P}, \tag{4.1}$$

i.e. as the ratio between the solution time on a single processor divided by the solution time on P processors. Ideally, we would like S_P to be equal to P, implying the P processors should be able to solve the problem P times as fast as a single processor. A more realistic situation is depicted in Figure 4.1: for small systems (i.e. when P is small), we typically get good speedup for a fixed problem. As more processors are added, the computational task per processor is reduced, while the communication overhead between the processors typically increases. Hence, after a certain number of processors, it does not pay off to add more processors.

Note that the above situation gives a too pessimistic view of supercomputing. The assumption about a fixed problem size is typically not correct. With the availability of larger computing systems, the problem size is typically also increased. Many problems solved on large systems are of a size that cannot be solved in a single processor context, either because of the storage requirement, because of the computational cost, or both. From this view point, the definition of speedup in (4.1) must be used with some care.

Finally, a couple of important reminders: when working in a multiprocessor environment, the single-processor performance is still of utmost importance. In particular we recall some of the possibilities of parallelism within a single processor: bit-level, instruction-level, pipelining, and superscalar capabilities (or multiple floating point units).

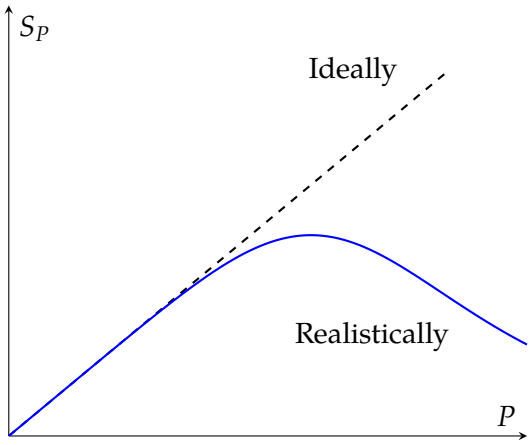


Figure 4.1: Ideal speedup ($S_P = P$) and realistic speedup.

4.2 Organization of multiprocessor systems

According to Almasi and Gottlieb (1989), a parallel computer is a collection of processing elements which communicate and cooperate to solve a problem fast. This definition raises some immediate questions:

- How many processing elements should we use?
- How should the processing elements communicate?
- Will the parallel computer be scalable?
- How powerful should a processor be?
- What about programming?

Figures 4.2 and 4.3 show two examples of organizations. In Figure 4.2 each processor has a local cache and can access memory modules via some type of interconnect. In this case, all the memory modules can be accessed directly by all the processors. This organization is referred to as *global* or *shared memory access*. In Figure 4.3 each processor has local memory and can only access information in other memory modules via an interconnecting network. This organization is referred to as *distributed memory access*. Each organization has its strengths and weaknesses. We will return to some of these issues later.

Uniform memory access

There are several ways to achieve global memory access. One type of systems is referred to as SMP: *Symmetric Multi-Processor*. This is a configuration where all the memory locations are "equidistant" in the sense that the memory access time is the same.

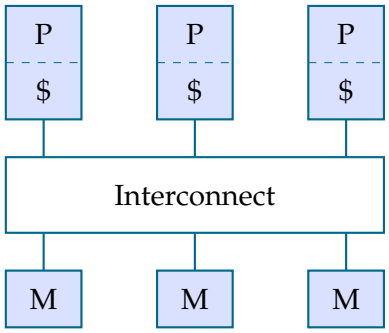


Figure 4.2: A parallel computer with global memory access.

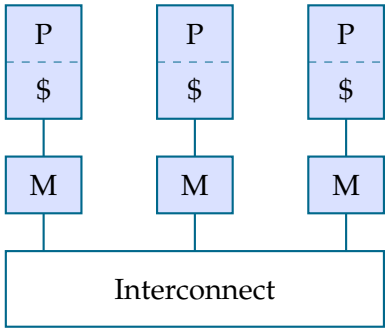


Figure 4.3: A parallel computer with distributed memory access.

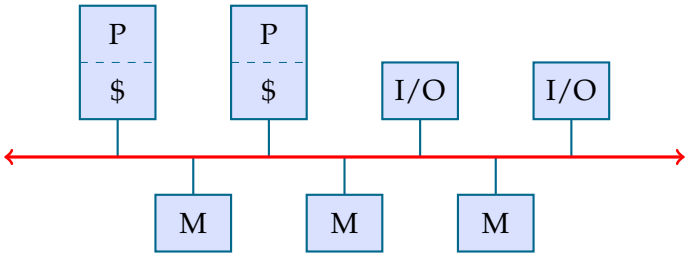


Figure 4.4: A bus-based organization. The global memory is accessible to each processor.

An example of an SMP is a bus-based organization; see Figure 4.4. This system has a broadcast interconnect similar to ethernet. It is inexpensive and it is easy to add processors. The disadvantage with a bus organization is the very limited scalability, which is due to the fact that the aggregate bandwidth is fixed. The use of caches could potentially alleviate some of this problem, but then the question of how to achieve cache coherency (or consistency) arises.

Another example of an SMP is a crossbar or a switch-based organization. Similar to a bus-based system, cache coherency is needed (e.g. via broadcast).

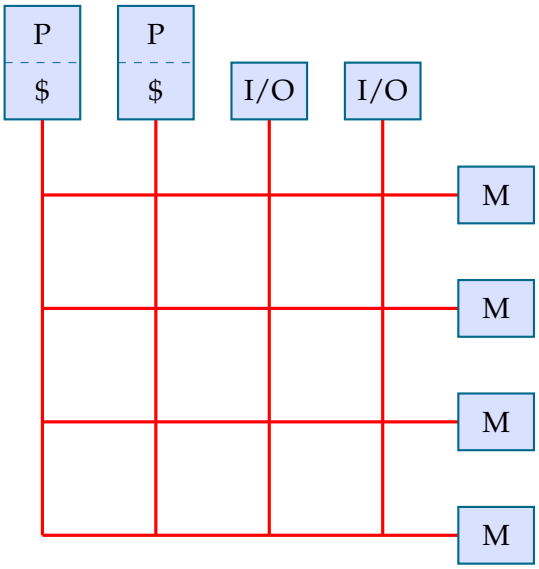


Figure 4.5: A crossbar interconnect. The global memory is accessible to each processor.

A crossbar has a better scalability than a bus organization. The aggregate bandwidth is increased, but adding a processor becomes more and more expensive as the system size grows (because the number of parts increases).

In summary, there is a scalability problem with the interconnect both with a bus organization and with a crossbar. In the case of the bus, the scalability issue is due to the fixed aggregate bandwidth; in the case of the crossbar, the scalability relates to the cost. An example of a compromise between these two issues is the multistage interconnect; see Figure 4.6.

Non-uniform memory access

An alternative to an SMP-organization is a NUMA-organization (*Non-Uniform Memory Access*); see Figure 4.7. The last supercomputers at NTNU all represent examples of this type of organization.

The Cray T3E had caches which were only used to hold data and instructions from local memory. The computer had no mechanism to keep the caches consistent with the global address space. The computer was an example of a non-coherent shared memory machine. In principle, any processor could read/write from/to any memory location in the global address space. In practice, however, messages were used to transfer data from one local memory module to another. The programming model was thus based on message passing. We will return to this model later.

On the SGI Origin, data from any memory location could be replicated into any of the caches. Hardware support existed to keep the caches consistent.

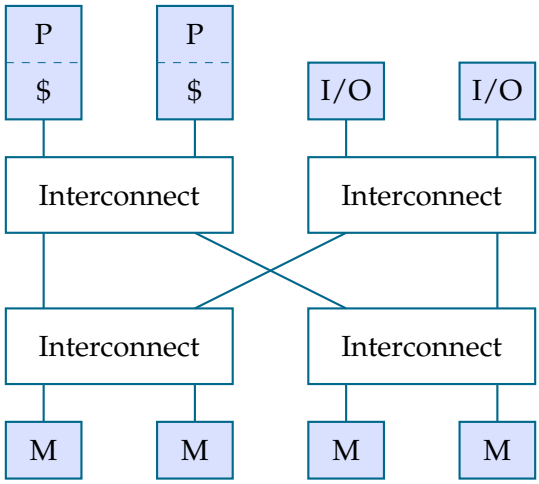


Figure 4.6: A multi-stage interconnect. The global memory is accessible to each processor.

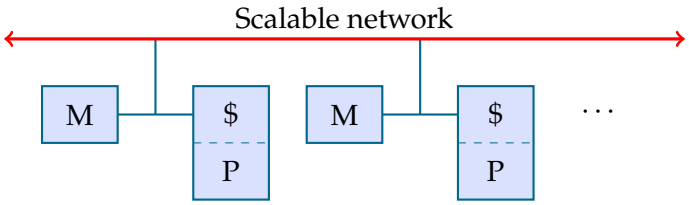


Figure 4.7: A NUMA organization.

This is also referred to as a ccNUMA organization (*cache coherent Non-Uniform Memory Access*). Any processor could read/write from/to any memory location. This feature could be exploited in a shared memory programming model. However, note that a message passing programming model could still be used on the SGI. We will return to a discussion of the current supercomputer at NTNU later.

Distributed memory access

An example of a distributed memory organization is the mesh interconnect as depicted in Figure 4.8. Each processor can access its own local memory similar to the single processor case. However, data from the local memory associated with the neighboring processors are obtained by message passing. The two first bits of the message are used to determine in which direction (North, South, East, West) to move. The two bits are then stripped off, and the message proceeds to the next step on the two-dimensional mesh. The message itself is trailing behind while the connection is being established. This approach is referred to as *wormhole routing* and was developed by William

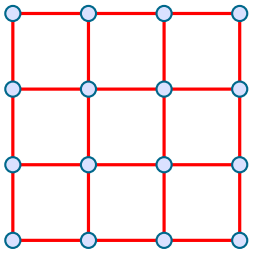


Figure 4.8: A 2D mesh interconnect. Each circle represents a processing element: a CPU, local cache, and a local memory, i.e., a single processor. Such a processing element is also referred to as a node.

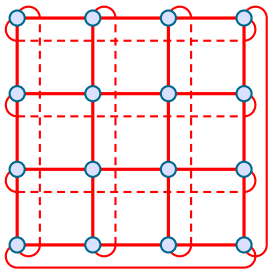


Figure 4.9: A 2D toroid interconnect.

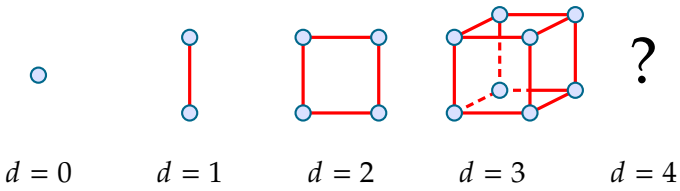


Figure 4.10: A d-dimensional hypercube has 2^d nodes (or processing elements), and the longest "distance" (number of hops) between any two nodes is d.

Dally at MIT. The mesh interconnect was used on the Intel Paragon and the Intel Delta supercomputers (in the 90s).

An improved version of the mesh interconnect is the 2D toroid; see Figure 4.9. Compared to the 2D mesh, there are wrap-around connections in each spatial direction (horizontally and vertically) in order to reduce the worst-case hop count.

Further extensions of the mesh interconnect are the 3D mesh or 3D toroid network topology (e.g. used on the Cray T3D and Cray T3E supercomputers).

Finally, we mention the hypercube organization; see Figure 4.10. This type of interconnect was attractive before the wormhole-routing. The whole message was typically passed through one hop before the next hop could start, something which resulted in very long communication times.

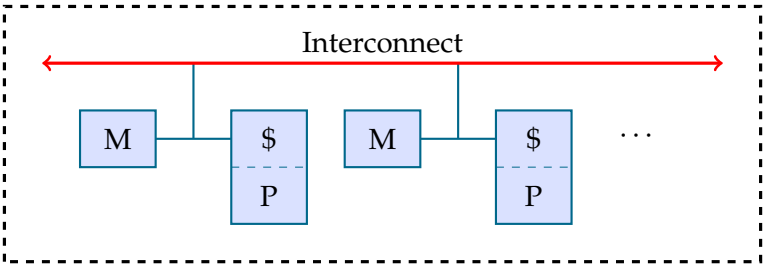


Figure 4.11: A Symmetric Multi-Processor (SMP)—sometimes also referred to as a node. A node typically comprises 16–64 processors with a cache-coherent uniform memory.

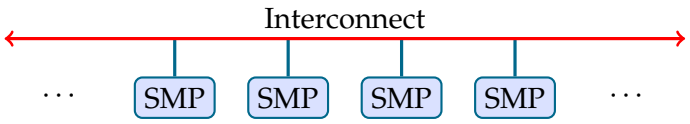


Figure 4.12: A ccNUMA supercomputer may comprise several SMP nodes, all linked together by an interconnecting network. All the caches are kept coherent, however, the memory access is only uniform within a single SMP; the memory access time varies between different SMPs.

MIMD computers

In the following, we will focus on MIMD architectures. MIMD is an acronym for *Multiple Instructions Multiple Data*, and refers to the fact that there are multiple instruction streams (one per processor or node) and multiple data streams (one per processor or node). A MIMD supercomputer is a general multiprocessor.

Alternative architectures are SISD - *Single Instruction Single Data* (a classical workstation), and SIMD - *Single Instruction Multiple Data* (sometimes called an array processor or a vector processor). An example of a SIMD interface is HPF: High Performance Fortran.

There are different types of MIMD computers:

1. MIMD with shared uniform memory (SMP, UMA), or MIMD with shared nonuniform memory (NUMA, ccNUMA);

2. MIMD with distributed memory and message passing.

Examples of the first category are depicted in Figures 4.11 and 4.12. In both cases, the global address space is available to every processor; hence, the term shared memory. A programming model for shared memory machines exists, and the current de facto standard is OpenMP; see http://openmp.org.

For the second class of MIMD computers, only local address space is directly accessible to each processor; see Figure 4.13. Access to non-local

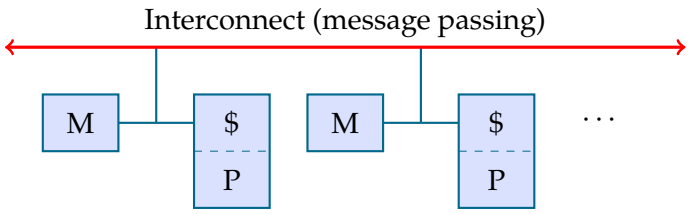


Figure 4.13: A distributed memory computer. Only the local address space is accessible to each processor. Different processors can only communicate via explicit message passing.

memory is achieved via message passing. The standard communication library used for message passing is MPI (*Message Passing Interface*).

We remark that programs written for distributed memory machines often run very well on shared memory machines. This is the reason why we will focus on a distributed memory programming model in this course, even though we may be running our programs on a shared memory machine. We thus make a distinction between the *programming model* we choose and the particular *machine* we use. On the other hand, if we had chosen a shared memory programming model, we could only use the program on a shared memory machine.

4.3 The current supercomputer at NTNU

We now give a brief overview of the current supercomputer at NTNU, a SGI (Altix) supercomputer based on the Intel i7 Sandy bridge microprocessor called *Vilje*.

The Sandy Bridge microprocessor

Multi-core systems is a recent trend in chip design. The i7 microprocessor represents an example of an octa-core chip. This means that a single chip integrates 8 processors (or cores) into a single integrated circuit; see Figure 4.14. A single chip therefore represents 8 physical processors in the sense discussed earlier. Each core is hyperthreaded meaning they can handle two instruction streams simultaneously. Thus, each chip has 8 *physical* cores but 16 *logical* cores. The idea behind this design is to attempt to hide memory latencies in the system by having one instruction stream do calculations, while the other stream is waiting for data, but there are no extra resources for calculation available. If your code is already efficiently supplying data to one of the instruction streams, the use of hyperthreading might actually harm performance.

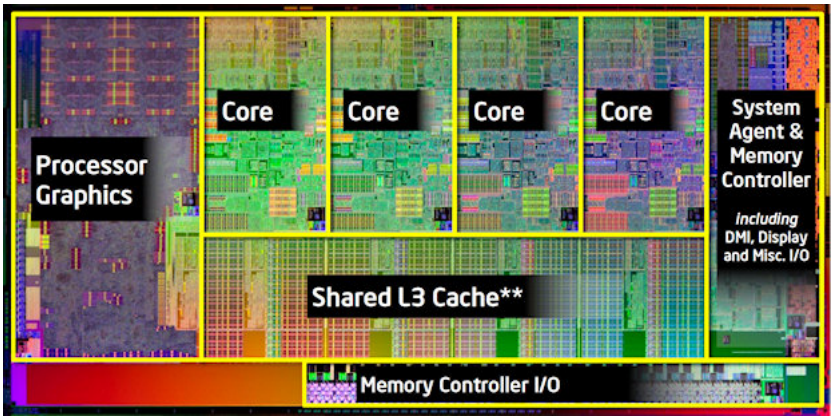


Figure 4.14: The figure depicts a quad-core Sandy Bridge microprocessor (or chip). The author was unable to locate an image of an octa-core chip.

Interconnecting chips to form larger SMPs

Multiple Sandy Bridge chips can be connected to form larger modules in SMP configurations. On Vilje, two chips are run in SMP mode on each compute node. Thus, up to 16 (32) processors share memory.

Interconnecting nodes to form a distributed SMP

Multiple nodes may also be connected to form clusters or distributed SMPs. In this case, data from one node to another must be passed across a high bandwidth low-latency switch network. A system based on multiple nodes must therefore be treated as a distributed memory system (at least globally).

Key data for Vilje

Vilje is based on 1404 nodes. The total number of processors (or cores) is therefore 22464.

Each node represents a shared memory system with 2 octa-core Sandy Bridge chips which share 32 GB memory (although a few nodes have 128 GB memory).

Each i7 processor operates at a clock rate of 2.6 GHz. The size of the L1 cache (3 clocks) is 32 kbyte for data and 32 kbyte for instructions. The size of the L2 cache (8 clocks) is 256 kbyte, while the size of the shared off-chip L3 cache is 20 Mbyte.

4.4 Programming models

We have seen that a node (with 16 physical processors) on Vilje represents a shared memory system where the aggregate memory for the node is globally available to all the 16 processors. A shared memory programming model (i.e. OpenMP) can therefore be used within a single node.

If we want to develop programs which can run on more than one node, we need to use message-passing (i.e. MPI). This is also the programming model we will emphasize in this course. Even though each node represents a shared memory system, the message-passing programming model may still be used within a node. However, the opposite is not true: a shared programming model cannot be used on a "pure" distributed memory system (e.g. on a PC cluster).

Note that a system like Vilje, which represents a shared memory system within a node, and a distributed memory system across the nodes, can also be programmed using both programming models within a single program.

4.5 Message passing

Message passing is fundamentally processor-to-processor communication. Only a local, unique memory is directly available to each processor. Both local and remote processes must cooperate in order to exchange data and/or synchronize (at least originally—some changes have been made in the extended version MPI-2).

Note that message-passing is a good way to use distributed shared memory machines (ccNUMA) because it provides a way to express memory/data locality.

Some of the key advantages of the message-passing model are:

- *Portability*: the model can be used on a collection of homogeneous or heterogeneous processors connected by a fast or a slow communication network;
- *Performance*: the approach exploits data locality, as well as the availability of a large, aggregate memory;
- *Expressiveness*: a limited communcation library suffices for most applications.

The Message Passing Interface (MPI) is the de facto standard for message passing. It is a *library*, not a language. MPI provides efficiency, portability and functionality. It represents a standardized communcation library running on a vast number of machines and architectures. Figure 4.15 illustrates the message passing model.

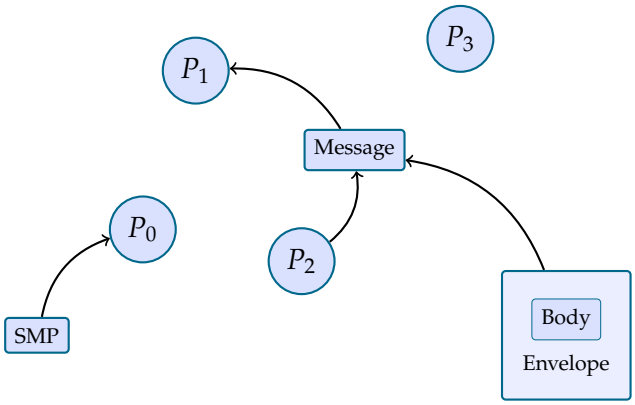


Figure 4.15: The message passing model. A number of processes, $P_0, P_1, \ldots, P_{n-1}$, are coupled together via a fast or a slow communication network. Each process has a local and unique memory/cache, and each process is associated with a particular computational task. The individual processes must communicate via explicit message passing. A message consists of an "envelope" which contains sufficient information about whether and when to open the message, as well as information regarding how to interpret the "body" of the message (the actual data). Note that the message is the only means of exchanging data between the processes and/or syncronizing the processes.

The original (1994) MPI-library represents the message passing model where both the local and remote processes cooperate e.g. via a send and receive operation. MPI-2 represents an extension of MPI where features like one-sided messages, parallel I/O, etc. are included.

The MPI operations can be classified in a few types of operations:

- one-to-one;
- one-to-all;
- all-to-one;
- all-to-all.

The first type is also referred to as point-to-point operations (send and receive), while the last three types are collective operations.

When we here talk about "all", we generally mean all processes $P_0, P_1, \ldots, P_{n-1}$ within a group of n processes. Such a group defines a *communicator* and the particular process number is referred to as the *rank* within that communicator. The default is to let all processes be members of the same (default) communicator. However, it is also possible to have some of the processes be members of one communicator (or group), while others be members of a

different communicator. In this context, "all" means all the processes *within* a particular communicator.

Finally, the collective operations can be further broken down into the following categories:

- data movement (broadcast; gather/scatter);
- collective computation (max/min; sum; etc.).

We will later explain in more detail the various MPI operations. A good way to learn MPI is by implementing a few simple examples. The whole library contains about 125 functions. However, as few as 6 may suffice for some problems. You only need to learn the functions needed for your particular problem. You may not have to learn the details of the whole library even for advanced applications.

An example

We now discuss a brief example of a program where the MPI library is used. The program listed below does the following: processor 0 sends a text message "Hello, world" to all the other processors. The other processors receive the message and all processors print out the message together with the their own process number.

Listings 4.16 and 4.17 show how this is done in both C and Fortran.

The output of the C version on the SGI Origin using $P = 4$ and $P = 8$ processors is shown in Listings 4.18 and 4.19.

Let us now comment on some of the statements here. We start with

```
#include <stdio.h>
#include "mpi.h"
```

The first statement is just a standard statement about including the header file associated with string (character) operations and I/O. The second statement is required if we want to use the MPI library in our program. In C, we need the statement

```
#include "mpi.h"
```

The equivalent statement in Fortran is

```
include 'mpi.f'
```

The MPI header file provides basic MPI definitions and MPI data types.

The first MPI statement needs to be:

```
MPI_Init(&argc, &argv);
```

This statement should only be called once. The last MPI statement is always

```
MPI_Finalize();
```

Listing 4.16: Hello world MPI in C.

```c
#include <stdio.h>
#include "mpi.h"

int main(int argc, char **argv)
{
  int rank, size, tag, i;
  MPI_Status status;
  char message[20];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  tag = 100;

  if (rank == 0) {
    strcpy(message, "Hello, world");
    for (i = 1; i < size; i++) {
      MPI_Send(message, 13, MPI_CHAR, i,
               tag, MPI_COMM_WORLD);
    }
  }
  else {
    MPI_Recv(message, 13, MPI_CHAR, 0,
             tag, MPI_COMM_WORLD, &status);
  }

  printf("node %d: %13s\n", rank, message);

  MPI_Finalize();

  return 0;
}
```

Listing 4.17: Hello world MPI in Fortran.

```
program hello
include 'mpif.h'

integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)
character(12) message

call MPI_INIT(ierror);
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror);
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror);

tag = 100;

if (rank .eq. 0) then
  message = 'Hello, world'
  do i=1,size-1
    call MPI_SEND(message, 12, MPI_CHARACTER, i, tag,
                  MPI_COMM_WORLD, ierror)
  enddo
else
  call MPI_RECV(message, 12 MPI_CHARACTER, 0, tag
                MPI_COMM_WORLD, status, ierror)
endif

print*, 'node', rank, ':', message

call MPI_Finalize(ierror)
end
```

Listing 4.18: Hello world MPI in C: 4 processors.

```
node 1: Hello, world
node 3: Hello, world
node 2: Hello, world
```

Listing 4.19: Hello world MPI in C: 8 processors.

```
node 5: Hello, world
node 1: Hello, world
node 7: Hello, world
node 2: Hello, world
node 3: Hello, world
node 6: Hello, world
node 4: Hello, world
```

This statement does not have to be the very last statement in your program, but it needs to be the last MPI statement. It statement ensures a clean exit.

Note that the command line arguments are passed to the C version of `MPI_Init`. The corresponding Fortran version reads:

```
call MPI_INIT(ierror);
```

Similar to a standard subroutine call in Fortran, a call to an MPI operation also starts with `call`. The parameter `ierror` is an integer and will return an error code in case something goes wrong. The C version also returns an error code. Instead of the statement used in the program listed above, we could alternatively have written

```
errorcode = MPI_Init(&argc, &argv);
```

In this case, the variable `errorcode` (an integer) will contain an error code if something goes wrong.

In general, any MPI statement in C has the format

```
errorcode = MPI_Xxxxx(parameters...);
```

or simply

```
MPI_Xxxxx(parameters...);
```

The particular MPI operation is given by "Xxxxx" where the name of the operation always starts with a capital letter and the remaining letters are lower-case. The number and type of parameters vary from operation to operation. For example, `MPI_Finalize` does not have any parameters at all.

In Fortran, any MPI statement has the format.

```
call MPI_XXXXX(parameters..., ierror)
```

where the parameter *ierror* returns an error code. Note that the name of the particular MPI operation is always in capital letters.

The next two statements after the initialization of MPI are:

```
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
```

The first of these statements returns the total number of MPI processes, while the second one returns the individual process number (the "rank"). More precisely, the process number is stored in the location pointed to by the second argument. `MPI_COMM_WORLD` is the default name of the communicator (the "univertse") and which include all the processes. This can be changed in order to create several separate "universes."

Note that the program listed in this example runs separately and independently on every processor on a multiprocessor. We also refer to this as SPMD - *Single Program Multiple Data*. All the problems we will study in this course will be of this type: the program running on each processor will be the same for all the processors. However, the data each processor will operate on will typically be different. The synchronization of the program will be implicit via the MPI operations. We will return to this issue later.

When this program runs on a particular processor, the program does not automatically know how many other processors are involved; this issue is taken care of by the MPI operation `MPI_Comm_size`. In the multiprocess context, the program running on an individual processor does not automatically know what its associated process number is (who am I?); this issue is taken care of by the MPI operation `MPI_Comm_rank`.

Let us now proceed to the statements

```
if (rank == 0) {
  strcpy (message, "Hello, world");
```

The if-statement will only be true for one of the processes, namely, the process with process number 0. This is also referred to as the "root" process. On the root process, the string "Hello, world" is copied into the string variable `message`.

For all the other processes, the if-statement will not be true, and the program execution will move on to the MPI statement `MPI_Recv`. This means that all of the other processes will be waiting for the root process to send them data. The root process sends data to the other processes in the loop

```
for (i=1; i < size; i++) {
  MPI_Send(message, 13, MPI_CHAR, i,
           tag, MPI_COMM_WORLD);
}
```

Several comments are in order here. First, note that the root process sends out the same message (string) to all the other processes. This is done in a loop. However, note that this loop corresponds to a *sequential* execution meaning that a message will be sent to process 1 before process 2 etc.

Let us now discuss the particular format in the parameter list for the operation `MPI_Send`. The general format for this operation is

```
MPI_Send(start, count, datatype, dest, tag, comm);
```

The first three parameters (start, count, datatype) represent *the data*, while the last three parameters (dest, tag, comm) represent *the envelope*; see Figure 4.15.

We now explain all these parameters in some more detail.

- *start*: initial address of the send buffer
- *count*: number of elements sent (of type `datatype`)
- *datatype*: e.g. `MPI_INT`, `MPI_FLOAT`, `MPI_CHAR` etc.
- *dest*: destination process (integer)
- *tag*: integer message identifier (e.g., `MPI_ANY_TAG`)
- *comm*: an ordered group of communication processes (same for send and receive)

Only the root process sends a message. All the other processes are waiting to receive a message. This is expressed by the MPI operation `MPI_Recv`. The general format for this operation is

```
MPI_Recv(start, count, datatype, source, tag, comm, &status);
```

The first three parameters (start, count, datatype) represent *the data*, while the last three parameters (dest, tag, comm, status) represent *the envelope*; again, see Figure 4.15.

Most of the parameters are similar to the MPI_Send operation:

- *start*: initial address of the receive buffer
- *count*: number of elements sent (of type *datatype*)
- *datatype*: e.g. `MPI_INT`, `MPI_FLOAT`, `MPI_CHAR` etc.
- *source*: source process (integer), i.e., the process sending the message
- *tag*: integer message identifier (e.g., `MPI_ANY_TAG`)
- *comm*: an ordered group of communication processes (same for send and receive)
- *status*: a structure providing information on the completed communication

Note that in the small example program, an explicit source is given, namely, the root process. Alternatively, we could have used `MPI_ANY_SOURCE` since each process in receive mode only expects one message. Similarly, we note that the "tag" is explicitly given. Alternatively, we could have used `MPI_ANY_TAG` since this parameter is not critical in our case.

One additional comment regarding the send and receive statements. This is an example of point-to-point communication. There exists several versions of send and receive. The type used here is called *blocking* send and receive. This means that the program running on the root processor cannot proceed until each message has been safely sent and the send buffer can be safely used again. Similarly, the program running on each of the other processors cannot proceed until the expected message has been received in the receive buffer.

Let us now comment on the parameter *count* used in the send and receive statements. In the C version, the count parameter is set equal to 13, while it is 12 in the Fortran version. The reason for this is that the `strcpy` function in C will append `\0` (the null character) at the end of the message. This is a symbol which indicates the end of the string. Even though "Hello, world" comprises 12 letters (one byte per letter), the message buffer in C requires 13 bytes of memory.

The parameter "datatype" in the send and receive operations has to be the same. In this case, the type is `MPI_CHAR` (in Fortran the corresponding data type is called `MPI_CHARACTER`). This is one of several predefined data types. Other important ones include `MPI_DOUBLE` and `MPI_INT`.

Finally, note that the output from the programs is not in a sequential order. The order may also change if we run the program over again.

Chapter 5

Shared memory machines

5.1 Introduction

Thus far in the course we have considered programming parallel machines using a distributed memory model, where several processes communicate through message passing, using the MPI library. This has been the traditional programming model used in the HPC community since distributed memory machines became popular during the late eighties. One of the main benefits of this programming model is that it can be used on all hardware, even machines that actually have a shared memory architecture. However, it has one major drawback; parallelizing a code typically requires substantial changes to the serial version.

In these notes we consider programming shared memory machines using a shared memory programming model. In particular we consider parallelization using the *OpenMP* standard. In later years, technology have reached a point where making a single processing core much faster is very challenging. In order to keep up with Moore's law for the next decades, vendors have started integrating several processing cores in one chip, even in processors targeted at desktop computers. At the moment, dual core processors are the norm on the desktop, with quad and hexacore processors being available in the high end market. All major vendors have signaled that they expect 8-16 cores to be the norm within a few years, and a few hundreds within a decade. This means that even for desktop class programs, programmers need to start developing parallel programs to utilize the available computing resources. The effect of this is a substantially increased interest in developing programming tools which allows for code parallization on shared memory archictures with little effort. While the OpenMP standard, which originated in the late ninties, initially was aimed at HPC applications, it has now been adapted much more broadly. The effort to improve the standard for the desktop market has also largely benefited the HPC community. The increased activity has resulted in two major revisions of the standard in the last five

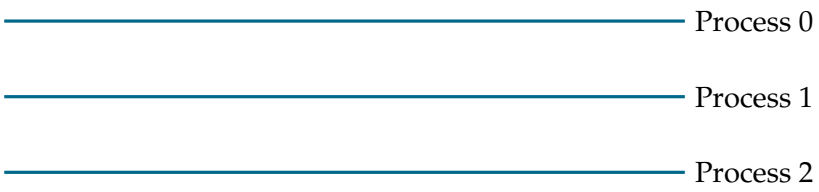


Figure 5.1: An illustration of the MPI programming model. We have several independent processes, and each of these processes have their own separate program flow.

years, making it quite extensive. The scope of this document is not to give a thorough introduction to the whole API, but rather to give an idea of the main principles, as well as what new challenges and benefits programming using OpenMP offers compared to the traditional message passing tools.

5.2 The OpenMP programming model

OpenMP is a C/Fortran language extension for programming shared memory parallel machines. It is implemented and supported by most major vendors, including Intel, AMD, IBM and Oracle (SUN). It is also available in open source compilers such as GCC or Clang, as well as in professional versions of Microsoft Visual Studio.

It is built on the threading paradigm in combination with a parallel section view of the code. This means that the main program flow happens on one processor only, which is quite a difference from the MPI programming model. Consider Figure 5.1, which is an illustration of the MPI programming model. Here we have several processes, and each of these processes have their own separate program flow. That is, each process is a separate instance of the program. Syncronization of these processes is typically handled implicitly by using blocking communication calls, i.e. if you call `MPI_Send` in one process, this process halts until it receives a confirmation that the message has been processed. Likewise, on the receiver end, the process blocks the program flow in the `MPI_Recv` call until it has received the expected message.

In the OpenMP programming model, this is different. Consider Figure 5.2 which is an illustration of this programming model. OpenMP is based on a fork/join programming model, where we only have a single instance of the program, i.e. the main program flow only happens on one processor. We mark certain sections of the program as being suitable for parallelization. When the program flow enters these sections of the code, the program *forks* into several threads which work independently of each other. At the end of the code sections the threads *join* with the thread running on processor 0 and the program flow is returned to this single processor. Thus in some sense one might say that the MPI programming model is embedded in the

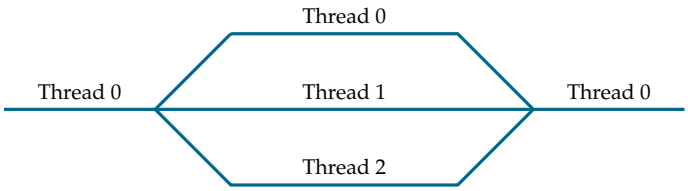


Figure 5.2: An illustration of the `OpenMP` programming model. We only have a single process, hence the main program flow only happens on a single processor. In sections of the code that we have marked as parallel, the program forks into multiple threads which work independently. At the end of the parallel section, these threads join together again and the program flow is returned to the first processor.

OpenMP model, but only in those parts of the code we have marked as parallel sections. This is only half the truth though. In MPI each process have their own *private* resources. Here however, this is not true. The threads within the parallel sections all have access to the *same* resources. This is crucial to keep in mind when designating the parallel sections of the code. One of the more common pitfalls is several threads trying to write to the same memory location, often due to using shared buffers during the calculations. It is thus highly recommended that you try to design your parallel sections in such a way that each thread has its own separate working buffers.

Critical section

If for some reason you cannot avoid several threads needing write access to the same resources, your only choice is to construct a critical section in your code to protect these resources. Consider Figure 5.3. A critical section of the code is a section of the program in which only a single thread can be at any point in time. We construct such sections of the code using a tool known as a *mutual exclusion lock*, commonly referred to as a *mutex*. We make a section of the code critical by embedding it in a lock/unlock procedure. Prior to entering the critical section of the code, the thread requests a lock of the mutex. If the mutex is open when this is requested, the mutex is locked, and info about which thread has the key is recorded. This mutex is now locked until the thread associated with the key requests an unlock. The thread then moves on executing the critical section of the code. Upon completion of the critical section the thread unlocks the mutex and continues doing whatever we have told it to do next. Now, if a second thread attempts to lock the mutex while it still is locked, the second thread will stall in the lock call until it is able to obtain the key. Since the mutex only has a single key, it would only be able to obtain this key after the mutex has been unlocked by the thread which is currently holding it. Since this unlocking only happens after the

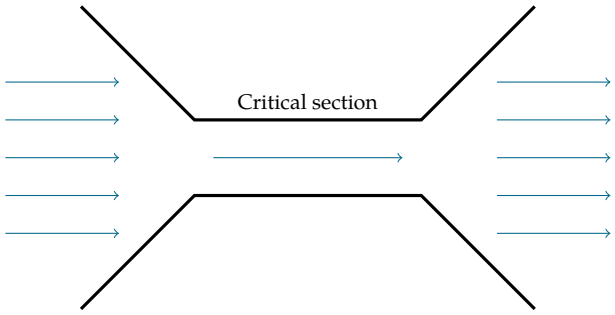


Figure 5.3: Illustration of a critical section. The incoming arrows represents the threads. All the threads are running concurrently, until they need to enter the critical section. Since only one thread can be inside the critical section at any time, the threads which want to enter need to wait until they obtain the key to the mutual exclusion lock, and hence their turn to enter the code section.

critical section has been executed, we can then guarantee that only a single thread is within the critical section of the code at any time. We stress that this is something you should only use if there is no way around it, since the use of a mutex leads to a *serialization* of the critical section code. This can be catastrophic for the parallel performance of your code if a large part of the computation time is spent within such critical sections. Since this is meant to be a brief introduction, we will not discuss these issues further in the following.

5.3 How to use OpenMP

The idea behind the OpenMP API is that we give the compiler instructions on which sections of the code we want to be parallelized. This means that, in contrast to MPI, which works with all compilers, OpenMP requires specific support in the compiler. The compiler then handles work division between the available number of threads. This is in stark contrast to MPI where work division is something the programmer always have to decide up front, before modifying the serial code accordingly.

These instructions to the compiler are known as *pragma* commands. There are mainly two classes of OpenMP pragmas. The first class of pragmas are those which can be used in combination with loop constructs such as *for* loops. This is the most useful case in context of HPC, since the programs typically consist of multiple loops which apply the same operation to large datasets. Consider the serial snippet

```
for (int i=0; i<100; i++)
  DoSomething(i);
```

or the equivalent in Fortran

```
do i = 1,100
  call DoSomething(i)
end do
```

For simplicity, we here assume that `DoSomething(i)` does not depend on any global resources, such as temporary working buffers. Hence this loop is highly suitable for parallelization using OpenMP. In addition we first assume that `DoSomething(i)` has a constant cost. To divide this loop among several threads, we simply do

```
#pragma omp parallel for schedule(static)
for (int i=0; i<100; i++)
  DoSomething(i);
```

Here we have our first example of an OpenMP directive. The pragma can be broken down into three parts.

`#pragma omp` all `OpenMP` directives start with this.

`parallel for` instructs the compiler that we want the following `for`-construct parallelized.

`schedule(static)` instructs the compiler to hand each thread approximately the same number of loop iterations up front. This is a good solution here since (we have assumed that) each call to `DoSomething(i)` has the same cost. Hence such a simple division will give good load balancing between the threads.

The ingredients in the Fortran version is the same, but the syntax is slightly different.

```
!OMP DO SCHEDULE(STATIC)
do i = 1,100
  call DoSomething(i)
end do
!OMP END DO
```

To avoid having to restate everything twice, we only give C examples in the following.

We can also run into situations where each call to `DoSomething(i)` has a different cost. One example where you can run into this scenario is if `DoSomething(i)` consists of an iterative method such as conjugate gradients. Each solution process can have a different solution time. In this case, if we give each thread a fixed number of loop iterations up front, we end up with poor load balancing between the threads. Fortunately OpenMP offers a mechanism to handle these situations. We simply do

```
#pragma omp parallel for schedule(dynamic)
for (int i=0; i<100; i++)
  DoSomething(i);
```

The only difference is the change of the schedule parameter in the pragma from static to dynamic. We here instruct the compiler to use a dynamic workload division between the threads. This means that we reserve one thread as a bookkeeper/negotiator. Within this parallel section this thread has a simple task; keep track of which loop iterations have been performed and hand out a new one to a thread when it requests it. Initially each thread is given a single loop iteration to perform. Once the thread finishes this, it asks the negotiator thread for a new one. The threads keep doing this until all work has been performed.

If `DoSomething(i)` is fairly costly, this works very well. However, in some cases each DoSomething(i) may be rather cheap. In this case the cost of asking the negotiator for a new loop iteration between every calculation may dominate the actual computation time. OpenMP also offers a mechanism to try to minimize this problem. Instead of having the negotiator hand out a single loop iteration when a thread asks for more work, it can hand out loop iterations in chunks. Consider

```
#pragma omp parallel for schedule(dynamic,5)
for (int i=0; i<100; i++)
  DoSomething(i);
```

The second parameter in the schedule (5) is the *chunk size*. This is the number of loop iterations a thread is (at most) given when it requests more work from the negotiator thread. Thus we can limit the number of times a thread has to communicate with the negotiator, hopefully making this part of the process less dominating.

OpenMP also offers a third scheduling mode, called *guided*. Consider

```
#pragma omp parallel for schedule(guided,5)
for (int i=0; i<100; ++i)
  DoSomething(i);
```

This is essentially a variant of dynamic scheduling, where we start out with a large chunk size. The chunks are then exponentially decreased until we reach a minimum, as specified in the chunk size. The idea here is that allocating large chunks initially is good for performance, since these will typically overlap fairly well. However, when the number of loop iterations left is small, we may end up in a situation where all the remaining loop iterations are allocated to a single thread. This is bad for performance since the other threads would then be left idle. By using progressively smaller chunks, the chance of this happening is reduced.

The second class of OpenMP directives are not tied to loop constructs. Instead they are to be used if we have sections of the code which are completely independent of each other. Consider the snippet

```
DoJob1();
DoJob2();
DoJob3();
```

If we are certain that these jobs are independent of each other, we can tell the compiler this fact, and ask for the different *sections* to be executed in parallel on several threads. We do

```
#pragma omp parallel sections
{
#pragma omp parallel section
  {
    DoJob1();
  }
#pragma omp parallel section
  {
    DoJob2();
  }
#pragma omp parallel section
  {
    DoJob3();
  }
}
```

Here each section of the code would be performed in a separate thread, before the program flow again returns to processor 0 once all sections have been completed. This directive is not as useful as those used in combination with loop constructs, in particular we cannot as easily utilize a large number of threads. The reason for this is fairly straight forward. In this example we can at most use three threads, since there are only three sections of code specified. It is often hard to find a large number of independent code sections to allow for a larger number of threads. Large, expensive loops, however, are typically present in most codes.

5.4 π — OpenMP style

We have previously calculated π in both serial and MPI codes. We can certainly use OpenMP for this as well. In the original serial code we have a loop

```
double integrate(double x0, double x1, int n,
                 function_t f)
```

```
{
  double h = (x1-x0)/n;
  double result = 0.f;
  for (int i=0; i<n; i++) {
    double x = x0 + (i+.5f)*h;
    result += h*f(x);
  }

  return result;
}
```

This is the loop where the main work happens, and is what we should focus our effort on. In this case, it is embarassingly simple to parallelize the loop since there are no dependencies between the loop iterations. We can simply hand a number of iterations to each thread, and then sum up the results afterwards. OpenMP makes this convenient through the reduction directive

```
double integrate(double x0, double x1, int n, function_t f)
{
  double h = (x1-x0)/n;
  double result=0.f;
#pragma omp parallel for schedule(static) reduction(+:result)
  for (int i=0;i<n;++i) {
    double x = x0 + (i+.5f)*h;
    result += h*f(x);
  }

  return result;
}
```

5.5 Compiling and running an OpenMP application

We have a source file called *openmp.c*, and we want to compile this with OpenMP directives enabled. On a standard Linux computer this can be achieved by using the `-fopenmp` directive, i.e.

```
gcc -O3 -o openmp -fopenmp -c openmp.c
```

while using the Intel compiler as we do on Kongull or Vilje, the directive is simply `-openmp`, i.e.

```
icc -O3 -o poisson -openmp -c poisson.c
```

As long as we do not use any OpenMP utility function calls, we can still compile this code into a completely serial code, simply by removing the compiler switches. The compiler then simply ignore the pragmas, which

makes the code look exactly as the serial code from its point of view. This is in stark contrast to a MPI version of the code where we would have to do substantial changes which makes the program dependent on the MPI libraries.

To run our program using for instance four threads we do

```
OMP_NUM_THREADS=4 ./openmp 2048
```

5.6 Final remarks

Modern supercomputers typically consist of multiple SMP nodes interconnected in a NUMA organization, see the lecture notes. Clusters also fall into the same category, since even the commodity processors used here have several cores integrated in their chips as discussed in the introduction. In particular, Vilje is an example of such a machine. Here each SMP node consists of 16 hyper-threaded processor cores. This means that an application which is parallelized using OpenMP can at most use 32 threads, although for floating point dominated programs, running only one thread per physical core is adviced. If more computing resources is needed, we have no choice but to resort to a distributed memory approach using MPI. The same applies to Kongull, except here each SMP has 12 cores, which are not hyperthreaded.

A very natural question to ask is whether or not the two approaches can be combined. The answer to this is yes. In fact this approach often allows us the best of both worlds. Fine-grain parallelism is often intricate to exploit using a distributed memory model, while the convenience offered by the shared memory model often makes it fairly trivial to express. In addition, it is not always easy to say up front where exploiting fine-grained parallelism actually will improve the performance of your program. Since the modifications to the program using the OpenMP approach is minimal, we do not have to invest much effort just to benchmark whether or not parallizing a particular loop improves performance.

Coarse-grain parallelism, however, is often fairly involved to exploit using a shared memory model, in particular due to the complications involved in protecting shared resources such as working buffers. This often lead to excessive memory usage or serialization of substantial parts of the code through usage of critical sections, see section 5.2. Using a distributed memory model, this problem is nonexistent. Each process have their own private resources which are inaccessible from the other processes. Here expressing coarse grain parallelism is often just a matter of adjusting the limits on some loops, as well as adding the appropriate library calls for data exchanges between the processes when such exchanges are needed.

Hence a program where we utilize a message passing based approach, i.e. MPI, to express the coarse grain parallelism, while utilizing OpenP pragmas

to express the fine grain parallelism within each MPI process allows use to use each approach for what they are best at, while avoiding their weak sides.

5.7 Further reading

You can find the official OpenMP homepage at http://openmp.org. This page is a great resource for those who are interested in more details. In addition to having the description of the standard, it also contains links to several books on subject, as well as discussion forums where you can ask questions. If a source of tutorials are to be suggested, we can recommend https://computing.llnl.gov/tutorials/openMP/.

Acknowledgements: Stephan Diederich helped with proof reading and gave some valuable input while this chapter was written. The chapter is written by Arne Morten Kvarving. Your assistance was greatly appreciated.

Chapter 6

Basic linear algebra performance

6.1 Introduction

Simulation-based science and technology require a rich set of numerical algorithms. For example, in the context of numerical solution of partial differential equations we have seen the need to solve linear systems of algebraic equations. Solution algorithms for linear system of equations may again be classified as direct methods or iterative methods. In either case, the solution algorithms rely on basic linear algebra operations. Most of the floating point operations in a typical simulation code are associated with such operations.

Because of the importance of basis linear algebra operations, a special library called BLAS (*Basic Linear Algebra Subroutines*) has been developed to deal with such operations. The BLAS library is again classified into three levels:

- Level 1 operations: vector-vector operations;
- Level 2 operations: matrix-vector operations;
- Level 3 operations: matrix-matrix operations.

Two examples of level 1 BLAS operations are

$$\boldsymbol{y} := a\boldsymbol{x} + \boldsymbol{y} \tag{6.1}$$

and

$$\sigma = \boldsymbol{x} \cdot \boldsymbol{y} = \boldsymbol{x}^{\mathsf{T}} \boldsymbol{y} \tag{6.2}$$

Operation (6.1) is called a *daxpy* operation. Here, the input $\boldsymbol{x}$ of length n is scaled with a constant a and added to the second input vector $\boldsymbol{y}$, which is also

of length n. The result is then stored back in the vector $\boldsymbol{y}$, i.e. the original values in $\boldsymbol{y}$ are overwritten.

Operation (6.2) represents a dot product (or inner product) which we have discussed extensively earlier in the course. Here, from the two input vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ (both of length n) we compute the scalar σ.

An example of a level 2 BLAS operation is the matrix-vector product

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} \tag{6.3}$$

Here, the output vector $\boldsymbol{y}$ (of length m) is computed from the given input matrix $\boldsymbol{A}$ (of dimension $m \times n$) and the given input vector $\boldsymbol{x}$ (of length n).

An example of a level 3 BLAS operation is the matrix-matrix product

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B} \tag{6.4}$$

Here, the matrix $\boldsymbol{X}$ (of dimension $m \times n$) is computed as the product of the matrix $\boldsymbol{A}$ (of dimension $m \times k$) and the matrix $\boldsymbol{B}$ (of dimension $k \times n$).

In this set of lecture notes, we will discuss the performance of operations 6.1, 6.2 and 6.4 on Vilje. We will primarily focus on the single-processor performance, but we will also consider the multi-processor performance using the BLAS implementation included in the MKL library developed at Intel.

In particular, we will discuss the performance as a function of:

- basic linear algebra operation;
- programming aspects;
- high level programming languages;
- exploiting multiple threads.

Key data for Vilje

The current supercomputer at NTNU, Vilje, is based on 1404 nodes. The total number of processors (or cores) is 22464 physical cores, each core being hyperthreaded, thus having 44928 logical cores.

Each node represents a shared memory system with 2 octave-core Sandy Bridge chips which share 32 GB memory (although a few nodes have 128 GB memory).

Each processor operates at a clock rate of 2.6 GHz. The size of the private L1 cache is 32 kbyte for data and 32 kbyte for instructions. The size of the private L2 cache is 256 kB, while the size of the off-chip L3 cache is 20 Mbyte.

The latency associated with the different memory levels is: 8 clock cycles for L2, 30 clock cycles for L3, and 150 clock cycles for main memory.

Maximum theoretical performance

The maximum theoretical performance (or peak performance) is the maximum number of floating point operations completed per second. We may talk about the maximum theoretical performance for a single CPU, a single node, or for the entire machine.

The maximum theoretical performance of a single core of a Sandy Bridge chip is found as follows. First, each physical has a separate SIMD floating point unit (AVX). This is a superscalar/FMA capable (Fused Multiply and Add) vector unit which can operate on 4 double precision number simultaneously. This performance is achieved if the units can be filled fast enough with data, and after a certain start-up period (recall the earlier discussion about pipelining). With 1 AVX per physical core, the maximum theoretical performance per core is thus 8 floating point operations per clock cycle. With a clock cycle of 2.6 GHz, this translates into 21 Gflops per physical core. Note, since logical cores share AVX units, we cannot expect additional performance from using the hyperthreads, since in order to achieve this performance we are already using the full memory bandwidth of the machine.

Unfortunately, many operations only achieve a fraction of the maximum theoretical performance. The measured performance will typically depend on the the reuse of data (a high degree of reuse means less memory traffic) and how good the compiler is. For the basic linear algebra operations we will study here, we will see a large variation in performance. Not surprisingly, the specially developed BLAS library will typically give excellent performance. However, some of the observed differences may come as a surprise.

6.2 Compiling and running the programs

The source code and the buildsystem for the tests are found in the git repository at https://github.com/TheBB/TMA4280, in the code/performance directory. CMake is used to generate the different Fortran programs, and to generate a build system capable of building the testing suite on most machines, including Vilje or your local Linux/OSX machine. There is also a example job script, a script to postprocess the results and (for those interested) the Octave/Matlab scripts used to generate the LaTeX code for the tables in this document.

6.3 Vector-vector operations

In this section we briefly discuss some performance results for an important vector-vector operation: the daxpy-operation (6.1), which named as such: **d**ouble precision **a**lpha $\boldsymbol{x}$ **p**lus $\boldsymbol{y}$.

$$\boldsymbol{y} := \alpha \boldsymbol{x} + \boldsymbol{y},$$

Table 6.1: Performance results (in MFlops) for a standard daxpy implementation in C (implemented as a single loop), compared with the performance using BLAS.

n	-00	-01	-02	-03	BLAS
10^2	45.92	63.69	82.82	68.37	0.10
10^3	227.13	363.00	457.12	434.61	0.81
10^4	403.55	1122.64	1217.60	1246.42	7.76
10^5	415.85	1300.58	1322.22	1338.98	79.71
10^6	379.52	418.88	413.65	413.65	341.52
10^7	376.78	420.55	413.65	420.36	341.52

does exactly n operations (additions) on $O(n)$ data, and needs to store n floating point numbers back into memory. All this memory traffic will severely influence the performance we can reach. In contrast to the matrix-matrix multiplication, only $O(1)$ floating point operations are needed per floating point number stored. There is thus very little "reuse" of data in vector-vector operations. From these general considerations we expect vector-vector operations to perform significantly worse than the matrix-matrix product multiplication.

Performance results for daxpy

Tables 6.1 and 6.2 show the performance results of the daxpy operation. In general, a standard implementation of the daxpy operation in C (i.e., a single loop), performs just fine. This is to be expected since this operation is utterly memory bandwidth bound and not much can be done wrong.

6.4 Matrix-matrix multiplication

Let us first consider the matrix-matrix multiplication (6.4). In general, if $\boldsymbol{A} \in \mathbb{R}^{m\times k}$, $\boldsymbol{B} \in \mathbb{R}^{k\times n}$ and $\boldsymbol{C} \in \mathbb{R}^{m\times n}$,

$$c_{ij} = \sum_{l=1}^{k} a_{il} b_{lj}, \qquad \forall\, i = 1, \ldots, m, \qquad \forall\, j = 1, \ldots, n.$$

Written out, this is

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{ik}b_{kj} \qquad \forall\, i = 1, \ldots, m, \qquad \forall\, j = 1, \ldots, n.$$

Mathematically, there are thus k multiplications and $k-1$ additions for each index i, j. The total number of operations is then

$$\mathcal{N}_{\text{op}} = mn\,(2k-1)\,.$$

Table 6.2: Performance results (in MFlops) for a standard daxpy implementation in Fortran (implemented as a single loop), compared with the performance using BLAS.

n	-O0	-O1	-O2	-O3	BLAS
10^2	0.07	0.07	0.08	0.07	0.10
10^3	0.89	0.83	0.67	0.69	0.80
10^4	9.37	7.23	8.17	7.30	7.75
10^5	70.96	66.18	70.65	87.33	79.70
10^6	353.27	356.28	385.78	402.52	341.51
10^7	513.54	514.07	514.52	512.66	341.51

For the special case where $A, B, C \in \mathbb{R}^{nxn}$, the total number of operations is

$$\mathcal{N}_{\text{op}} = n^2 (2n - 1) \approx 2n^3.$$

Hence, if m, n, k are of the same order (e.g., $m = k = n$),

$$\mathcal{N}_{\text{op}} = \mathcal{O}(n^3). \tag{6.5}$$

Triple-nested loop

In a numerical program, matrix multiplication can be implemented in several ways. The most straightforward method is in a triple-nested loop as follows (using C).

```
for(i=0; i<m; i++) {
  for(j=0; j<n; j++) {
    c[i][j] = 0.0;
    for(l=0; l<k; l++) {
      c[i][j] += a[i][l]*b[l][j];
    }
  }
}
```

Here, the terms are accumulated relative to an initialized value of zero. Counting the number of floating point operations in the inner-most loop, there are one multiplication and one addition. The total number of floating point operations for matrix multiplication then becomes $\mathcal{N}_{\text{op}} = 2mnk$ ($= 2n^3$ when $m = k = n$).

Loop unrolling

Loop unrolling helps to make the data flow and the data dependencies more explicit and prepare for good use of the floating point units. For example, when $k = 10$, the inner-most loop can be unrolled manually to get the following alternative version:

```
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    c[i][j] = a[i][0]*b[0][j]
            + a[i][1]*b[1][j]
            + a[i][2]*b[2][j]
            + a[i][3]*b[3][j]
            + a[i][4]*b[4][j]
            + a[i][5]*b[5][j]
            + a[i][6]*b[6][j]
            + a[i][7]*b[7][j]
            + a[i][8]*b[8][j]
            + a[i][9]*b[9][j];
  }
}
```

Here, all the terms are written out explicitly, following the mathematical definition. The computation of `c[i][j]` now takes one addition fewer.

Loop unrolling reveals independent operations that can be performed concurrently, while reducing the index-related overhead of the loop. This makes it possible to make better use of the pipelined, superscalar (vector) floating point units of the processor. At high enough optimization levels, the compiler will typically try to unroll loops where it sees this as beneficial.

Performance results

We now present performance measurements for the matrix-matrix multiplication. The source codes used are given in the appendix. In all the tests we perform, $m = k = n$ (i.e., we consider square matrices). Both a C version and a Fortran version of the matrix-matrix operation will be tested and compared with the performance using the `dgemm` routine from the BLAS library. We provide the full program listings, together with a description of how the programs were compiled and run on Vilje. For example, we will explore the effect of using different levels of compiler optimization.

Note that we may call the Level 3 BLAS routine `dgemm` from either Fortran or C. When we use this library routine, the optimization level or code language should not matter. As long as n is large enough, it is possible to obtain a high degree of peak performance.

Table 6.3: Standard mxm (triple-nested loop) using C, compared with BLAS.

n	`-O0`	`-O1`	`-O2`	`-O3`	BLAS
100	274.55	566.00	562.76	563.32	775.64
500	226.53	503.96	496.40	496.84	14323.68
1000	180.78	214.33	214.37	214.36	16786.90
1500	169.95	205.26	204.69	204.55	17808.83

Table 6.4: Standard $m \times m$ (triple-nested loop) using Fortran, compared with BLAS.

n	`-O0`	`-O1`	`-O2`	`-O3`	BLAS
100	247.37	1666.47	4800.47	7109.23	775.64
500	191.22	982.85	3375.70	7565.13	14323.68
1000	142.11	210.15	1356.46	7228.42	16786.90
1500	122.92	205.13	1357.79	7584.47	17808.83

Finally, note that the performance results we list in the following are approximate and may not be exactly reproducable. However, they represent typical results and the conclusions we arrive at should be valid.

The programs were run on 16 processors even though the code contains no communication between the processors. This gave 16 timing results per run.

These versions are single threaded; the difference is only the compiler optimization level.

As can be seen from Table 6.3, the compiler optimization level had really disappointing influence on the obtained performance—in fact, it actually reduced it in some cases. These results looks far from flattering for the C programming language. Fortunately, this can be overcome by using BLAS, in this case through MKL. The BLAS version of the program was invoked by calling e.g.

```
mpirun -np 16 timing-O3 1000 2
```

For the Fortran version of the programs, however, as can be seen from Table 6.4, the compiler optimization level greatly influences the obtained performance.

We see that the language utilized greatly influnces the performance of this operation. This can be attributed to the fact that, given proper memory

Table 6.5: "Manually" unrolled innermost loop (`mxm_unr`), C.

n	-O0	-O1	-O2	-O3
10	328.19	485.23	577.84	409.26

Table 6.6: "Manually" unrolled innermost (`mxm_unr`), Fortran.

n	-O0	-O1	-O2	-O3
10	189.17	292.61	674.69	621.29

Table 6.7: BLAS with 16 threads (SMP). All performance numbers are in MFlops.

n	BLAS (16 threads)
100	1463.80
500	66599
1000	100527
1500	101389

handling, the matrix times matrix operation is dominated by floating point operations. The BLAS and (and to some extent) Fortran realizations manage to keep the pipelines filled with data, while the C version does not seem to be able to keep the floating point units fully occupied.

Let us also compare these performance results with the manually unrolled innermost loop (which we have done for the specific case $n = 10$). The programs are compiled and linked as before. When we activate `mxm_unr`, we obtain the results in Tables 6.5 and 6.6.

We do not observe much performance increase for either case. It seems that this is a trick the compiler uses extensively on its own. Note that the Fortran version is still about 50% faster than the C version (for `-O3`).

We now try to use the parallel (multi-threaded version) of BLAS implemented in the MKL library.

From Table 6.7 we see that we obtain very impressive performance. These numbers should be compared with the maximum theoretical performance for the entire node (i.e., using 16 processors) which is 16×21 GFlops = 336 GFlops. Note that the cannot get more than 21 GFlops per core since this

number corresponds to the maximum performance of the available multiply-and-add units per core. We thus see that BLAS (using the SMP version of the `MKL` library) achieves close to 33% of the maximum theoretical performance over 16 processors. This is actually quite impressive since this was achieved only by adding a few compile and run flags; the program itself is unchanged from earlier. This might seem a bit disappointing, but it just shows the real bottleneck in the computer: memory bandwidth.

Finally, we recall the reason for the excellent performance of the BLAS library for the matrix-matrix multiplication operation: the mxm operation uses $O(n^2)$ data and $O(n^3)$ operations, implying that we need to do $O(n)$ floating point operations per single floating point number stored. Because of the significant "reuse" of data, there is a significant potential to hide the memory latency and keep the floating point units busy. In particular, BLAS is able to get close to optimal single-processor performance, and a standard triple loop in Fortran is able to get fairly close using two threads per core. Unfortunately, the C version is not able to keep the floating point units busy enough.

Combining Fortran and C

We recall that memory is allocated column-wise in Fortran and row-wise in C. In BLAS the Fortran convention is used. It is thus necessary to be careful when using two-dimensional arrays when calling `dgemm` (or other routines following the Fortran convention).

To ensure correctness, there are several ways to proceed. By switching indices when accessing arrays in C, a “pseduo” column-wise allocation is achieved. This is the approach we chose to use here (and in the rest of the course). Sometimes it may be possible to use a version of the library that accepts row-wise allocation, such as the CBLAS library. However, CBLAS is not as universally available, so we have chosen the approach that is most portable.

Chapter 7

The Poisson problem

7.1 The Poisson problem

The Poisson problem typically models a diffusion process, and is a very important model problem in science and engineering. This is related to the fact that the Poisson problem may constitute the whole, or more commonly, part of a mathematical model describing a physical system.

Numerical algorithms for solving partial differential equations often decouple a complex problem into subproblems, of which the Poisson problem is an important one.

We now give a few examples. We start by considering the Poisson equation

$$-\nabla^2 u = f \tag{7.1}$$

defined in a domain Ω.

A physical example where this type of equation represents the governing equation can be found in electrostatics. In this case, the differential forms for the electric field are

$$\nabla \cdot \boldsymbol{E} = 4\pi\rho, \tag{7.2}$$

$$\nabla \times \boldsymbol{E} = 0, \tag{7.3}$$

where ρ is the charge density. It follows that the electric field $\boldsymbol{E}$ can be expressed as the gradient of a scalar field ϕ, i.e., $\boldsymbol{E} = -\nabla\phi$. Hence,

$$\nabla \cdot \boldsymbol{E} = -\nabla \cdot \nabla\phi = -\nabla^2\phi = 4\pi\rho. \tag{7.4}$$

The Poisson equation (7.1) is an example of an elliptic partial differential equation. It is typically solved on a bounded domain Ω, in which case we need to specify boundary conditions for u on the domain boundary $\partial\Omega$, e.g., the potential function ϕ defined on $\partial\Omega$. Note that the potential ϕ at any point in the domain Ω will depend on the specified potential along the entire boundary $\partial\Omega$.

A similar example is the potential flow approximation in fluid mechanics. If the velocity field $\boldsymbol{U}$ is irrotational and incompressible, i.e.

$$\nabla \times \boldsymbol{U} = 0, \tag{7.5}$$

$$\nabla \cdot \boldsymbol{U} = 0, \tag{7.6}$$

it follows that $\boldsymbol{U} = \nabla\phi$, where ϕ represents a scalar velocity potential and satisfies the Laplace equation

$$\nabla^2\phi = 0. \tag{7.7}$$

A third example where the Poisson equation represents the governing equation is *steady* heat transfer. In this case, the Poisson equation represents energy conservation in differential form. This can readily be derived by noting that the net energy transfered out of an arbitrary domain Ω can be expressed as

$$\int_{\partial\Omega} \boldsymbol{q} \cdot \boldsymbol{n} \, \mathrm{d}S = \int_{\Omega} f \, \mathrm{d}\Omega \ , \tag{7.8}$$

where $\mathbf{q}$ represents the heat flux, $\mathbf{n}$ is the surface normal along the domain boundary $\partial\Omega$ and f represents a volumetric heat source. In short, Equation (7.8) says that the net energy out of the domain must equal the net heat generation inside the domain; see Figure 7.1. Using Gauss' divergence theorem, we can write

$$\int_{\partial\Omega} \boldsymbol{q} \cdot \boldsymbol{n} \, \mathrm{d}S = \int_{\Omega} \nabla \cdot \boldsymbol{q} \, \mathrm{d}\Omega = \int_{\Omega} f \, \mathrm{d}\Omega, \tag{7.9}$$

from which we obtain that

$$\nabla \cdot \boldsymbol{q} = f. \tag{7.10}$$

The most common *constitutive model* to use is Fourier's law, which states that the heat flux is proportional to the temperature gradient, i.e., $\boldsymbol{q} = -\kappa\nabla u$ with $\kappa > 0$. Substituting this relationship into (7.10) gives

$$-\nabla \cdot \kappa\nabla u = f \qquad \text{in } \Omega \tag{7.11}$$

to be solved for the temperature u.

In the case of constant thermal diffusivity κ, the governing equation reduces to the Poisson equation

$$-\kappa\nabla^2 u = f. \tag{7.12}$$

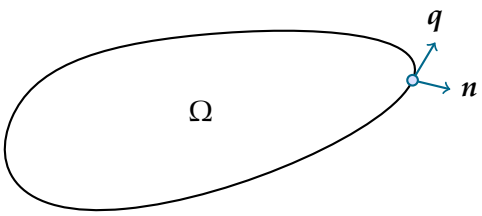


Figure 7.1: The domain Ω and a surface element on the boundary $\partial\Omega$. The outward unit normal vector $\boldsymbol{n}$ is indicated, together with the heat flux $\boldsymbol{q}$.

7.2 Unsteady Heat Transfer Problems

Assuming no fluid flow, energy conservation in the unsteady case is described by the the unsteady (parabolic) heat equation

$$\frac{\partial u}{\partial t} = \kappa \nabla^2 u + f \qquad \text{in } \Omega. \tag{7.13}$$

If we discretize this equation in time using the Euler Backward method, we obtain

$$\frac{u^{n+1} - u^n}{\Delta t} = \kappa \nabla^2 u^{n+1} + f^{n+1}, \tag{7.14}$$

where superscript n refers to a quantity at time t^n, $n = 0, 1, 2, \ldots$. This can also be expressed as

$$\left[-\kappa \nabla^2 + \frac{1}{\Delta t} \right] u^{n+1} = \frac{u^n}{\Delta t} + f^{n+1}. \tag{7.15}$$

Hence, a typical evolution problem discretized in time using implicit finite differences will necessitate the solution of a Helmholtz type equation at each time step. Note that the Helmholtz operator (the operator inside the parentheses) corresponds to the Laplace operator plus a multiple of the identity operator.

7.3 Eigenvalue problems

Eigenvalue calculations also form an important application area in science and engineering. A typical example is the computation of the first eigenmodes or eigenvibrations in a structure, e.g. a building, a bridge or a turbine. In order to avoid resonance phenomena in the structure, one can precompute the most important eigenmodes numerically, and design the structure such that resonance is avoided for a typical external load or excitation.

To illustrate a typical analysis, consider the following simple model problem:

$$-\nabla^2 u = \lambda u$$

with proper boundary conditions, e.g. the solution specified along the domain boundary. A numerical model is then constructed for this eigenvalue problem. This discretization will typically result in a large set of algebraic equations. The smallest eigenvalues/eigenmodes will give information of *physical significance*; for our specific model problem, they will approximate the eigenmodes for a diffusive system and the time constants associated with the decay of these.

7.4 Outputs from partial differential equations

In many engineering applications, the primary interest may not be the details of the solution u everywhere in the domain Ω, but rather some very specific output, e.g. the average temperature over part of the domain boundary, the drag on an underwater cable due to currents in the sea, or an eigenvalue (e.g. the lowest eigenfrequency). In these cases, the ultimate interest may just be a single number. However, in order to compute this output of interest, a numerical approximation u_h to u needs to be computed on the entire computational domain Ω, perhaps involving thousands or millions of unknowns.

7.5 Other important issues

There are many topics that are important in order to successfully obtain accurate numerical solutions for realistic physical problems.

Grid generation

In this course, we will primarily consider problems in one and two space dimensions. In one space dimension, the grid generation is trivial. However, for realistic two and three-dimensional domains, the grid generation itself may pose a major challenge. In the past, there has been much effort put into the construction of automatic mesh generators, however, this is still an ongoing research topic. It turns out that it is easier to decompose a general computational domain into triangles and tetrahedral elements than it is to decompose it into quadrilateral or hexahedral elements. This is the main reason why the use of triangular and tetrahedral (finite) elements tends to be fairly popular for representing general geometries; e.g. see Figure 7.2.

We will not focus on these issues in this course, and stick to simple, rectangular domains with cartesian meshes.

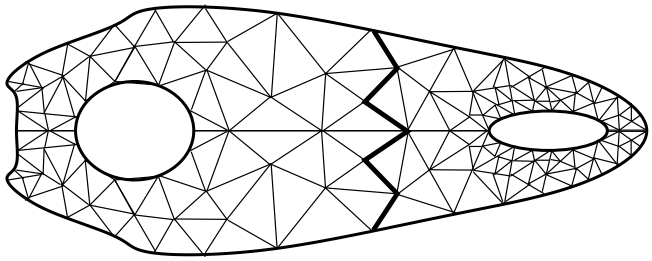


Figure 7.2: A triangulation of a two-dimensional domains and a partitioning of this domain into two subdomains (the bold line represents the subdomain interface).

Domain decomposition and parallel computing

A second important issue related to grid generation is *domain decomposition*. This is the decompostion of the computational grid into subdomains, where each subdomain again represents many degrees-of-freedom; see Figure 7.2.

Domain decomposition may be important for several reasons. If one is interested in using a parallel computer with a distributed memory, the global problem (including the grid) needs to be distributed among the processors. In this context, each individual subdomain may be associated with a single processor. However, even on a single process computer, domain decomposition may prove very useful in the construction of preconditioners for iterative solution methods, i.e. methods that will speed up the convergence rate when solving the system of algebraic equations.

Creating a good decomposition from a given grid is not always a trivial task for general meshes. This is also a field where much progress has been made over the past few years.

We will come back to this in the course, both in the context of solving the discretized equation using parallel computers as well as in the context of preconditioner construction.

Adaptivity

For complex problems, the initial grid is typically generated such that areas where one expects the solution to exhibit large gradients are well resolved (i.e. a higher density of elements or degrees-of-freedom is used in those areas). This approach is very heuristic, and the quality of the underlying mesh depends strongly on how well one is able to predict the solution structure.

An improved strategy is to use the initial (perhaps coarse) mesh to obtain a temporary solution which is subsequently used for the purpose of refining the mesh where the error is expected (or estimated) to be large. This is called *a posteriori* error estimation and *adaptive* grid refinement (or perhaps

unrefinement). It is an area of considerable research effort since it promises better error control and reduced computational cost for a fixed error target. However, adaptive grid refinement is not a trivial task to analyze or to implement, in particular, for unsteady problems where the solution structure may change as a function of time.

We will not focus on adaptivity in this course.

Visualization

Once a numerical solution has been computed, the computational results need to be interpreted. For a multi-dimensional problem, one typically ends up with a large amount of data. Visualization of these data is often essential in order to extract useful information from the simulation. Even though the final answer we are looking for may just be a single number (e.g. the drag), we often want to understand some of the main features in the solution. With huge amount of data, visualization (including feature extraction) are invaluable tools. In the context of parallel computing, visualization is particularly important due to the very high volume of raw data.

We will consider how to handle the output in this course, but will not focus on visualization tools. Matlab and Octave will suffice for our needs.

Software development

The effort needed to design and implement a software package for large-scale simulation of physical systems is typically significant. The associated cost is therefore also very high. Hence, it is very important to consider good ways to break down the global problem into smaller modules which can interact in a flexible and efficient manner.

The use of *object-oriented design* has become popular in recent years. One of the key issues when designing and implementing a large software package is to be able to identify commonalities between the various computational tasks, and to properly encapsulate these so that they may be made into more generic modules which can be reused for different purposes. An example of this is the solution of the Poisson equation, which may be used for multiple purposes in the same simulation package. Another key issue is the concept of data abstraction where one tries to implement higher-level functionality without necessarily having to worry about all the details in the lower-level tasks.

7.6 Final comments

It is common to consider the numerical solution of partial differential equations in the following conceptual model:

1. we know the computational domain Ω and the various parameters and input data (e.g. the thermal diffusivity κ and the volumetric heat source f);

2. we compute a discrete approximation over the entire domain;

3. we compute the actual output of interest and otherwise interpret and visualize the results.

The above approach is often referred to as a *forward problem*, and it may be sufficient for many problems. However, for certain applications, this mode of analysis and computation will not suffice. We may not always know the precise shape of the computational domain, e.g. an airplane wing. In fact, finding the shape which minimizes the drag may be part of the objective with the simulation. In such a case, many forward problems are solved, each one hopefully getting closer to an optimal solution. This type of application represents an example of an optimization problem. In this context, we note that computing the solution of a partial differential equation may only give a single data point in a larger optimization algorithm.

Another area where numerous forward problems may be required is for applications where the input parameters (e.g. the thermal diffusivity κ) are not known, but in fact the quantity of main interest. For such applications, one will typically have available a certain number of *measurements* corresponding to multiple *outputs* from the governing equation. The objective is then to find a distribution of the thermal diffusivity inside the computational domain such that the difference between the simulated outputs and the real, physical outputs (or measurements) is minimized. This type of application represents an example of an *inverse problem*, and is of significant importance in areas such as medical imaging, estimation of rock properties etc.

Chapter 8

Finite differences for Poisson

8.1 Discretization of equations

When we want to solve a partial differential equation on a computer, we can typically only do so in an approximate sense, since a computer can only deal with a finite amount of data. The process of turning a continuous equation into a finite-dimensional equation suitable for solving on a computer is referred to as *discretizing* an equation.

There are several ways to go about this, the most popular being *finite differences*, *finite elements* and *finite volume* discretizations. Common to all of these approaches is that at the end of the day, the partial differential equation is turned into a set of linear equations to solve, i.e. you end up with something on the form

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{g}$$

where $\boldsymbol{A}$ is the matrix of linear equations, $\boldsymbol{u}$ is the vector of unknowns we seek and $\boldsymbol{g}$ is the load (the right hand side in the equation system).

The simplest and least technical of these are the finite difference approach. Since this is not a course in numerical solution of partial differential equations, we will focus on this approach only in this course. But most of what we consider is also applicable to the other forms of discretization due to the fact that we will mostly focus on the solution of the linear system of equations.

8.2 Finite difference approximations

Consider the function $u(x)$ depicted in Figure 8.1. A grid has been introduced—that is, we only consider the function in a discrete sets of points, $\{x_i\}_{i=0}^{N}$, with $x_i = x_0 + ih$. Here h is the grid spacing, here taken as a constant, but in principle we can have a different spacing between each discrete grid point. We then estimate the derivatives of this function, only using the values of the function in the discrete sets of points. This approximation is called a *finite*

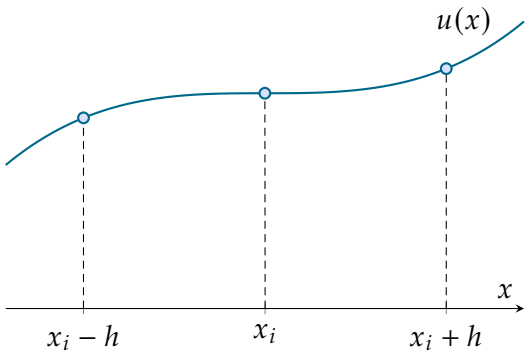


Figure 8.1: The function $u(x)$ sampled at a finite difference grid.

difference. We now give alternative finite difference approximations of $u'(x)$ and $u''(x)$ at $x = x_i$.

a forward difference approximation:

$$\frac{u(x_i + h) - u(x_i)}{h} = u'(x_i) + O(h);$$

two central difference approximations:

$$\frac{u(x_i + h) - u(x_i - h)}{2h} = u'(x_i) + O(h^2),$$

$$\frac{u(x_i + h) - 2u(x_i) + u(x_i - h)}{h^2} = u''(x_i) + O(h^2).$$

The forward difference approximation of $u'(x_i)$ is of first order, meaning that the error in approximating the first derivative scales linearly with h: if h is reduced by a factor of two, the error is reduced by a factor of two. The central difference approximations of $u'(x_i)$ and $u''(x_i)$ are of second order, meaning that the error in approximating the first and second derivatives scales quadratically with h: if h is reduced by a factor of two, the error is reduced by a factor of four.

We can also generate higher-order approximations to the first and second derivative of $u(x)$. Higher-order approximations will involve couplings between more neighboring points.

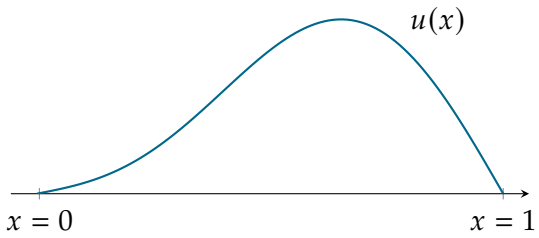


Figure 8.2: Domain and solution of the one-dimensional Poisson problem.

Figure 8.3: A finite difference grid.

8.3 The one-dimensional Poisson problem

Homogeneous Dirichlet boundary conditions

We consider here the Poisson equation (or diffusion equation) in one space dimension,

$$-u_{xx} = f \qquad \text{in } \Omega = (0, 1)$$

and with homogeneous boundary conditions,

$$u(0) = u(1) = 0.$$

In the following, we will denote the derivative of u with respect to x as u_x, and the second derivative of u with respect to x as u_{xx}. This will prove useful when we later consider two- and three-dimensional problems.

In the Poisson equation, the right hand side $f(x)$ is assumed to be known; $f(x)$ is often referred to as the source term. In the particular case when $f = 0$, the Poisson equation reduces to the Laplace equation.

In general, the Poisson equation is a partial differential equation (PDE), which in one space dimension reduces to a standard ordinary differential equation. In order to obtain a unique solution, we need to specify boundary conditions. In our case, u is specified at the end points $x = 0$ and $x = 1$. When u is specified on the boundary, we say that we have prescribed Dirichlet boundary conditions. When the prescribed values are zero, as in our case, we say that we have prescribed homogeneous Dirichlet boundary conditions. The Poisson equation together with the boundary conditions constitute the Poisson problem.

Let u_i be an approximation to $u(x_i)$, $i = 1, \ldots, n-1$, and let $f_i = f(x_i)$. A finite difference approximation of the Poisson problem can then be expressed

as

$$-\left(\frac{u_{i+1}-2u_i+u_{i-1}}{h^2}\right) = f_i, \qquad i = 1,\ldots,n-1, \tag{8.1}$$

$$u_0 = 0, \tag{8.2}$$

$$u_n = 0. \tag{8.3}$$

We have here $n-1$ unknown values to determine, namely, $u_1, u_2, \ldots, u_{n-1}$, and we have $n-1$ conditions by requiring that the Poisson equation be approximated at all the internal grid points $x_1, x_2, \ldots, x_{n-1}$. The values u_0 and u_n follow from satisfying the boundary conditions. Note that we have here used a second order finite difference approximation of u_{xx}.

The equations (8.1) can also be expressed as the system

$$\begin{aligned} 2u_1 - u_2 &= h^2 f_1, \\ -u_1 + 2u_2 - u_3 &= h^2 f_2, \\ &\vdots \\ -u_{n-2} + 2u_{n-1} &= h^2 f_{n-1}. \end{aligned}$$

We have here already used the fact that $u_0 = 0$ and $u_n = 0$.

In matrix form, this system can be expressed as

$$\underbrace{\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}}_{\boldsymbol{A}} \underbrace{\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{pmatrix}}_{\boldsymbol{u}} = h^2 \underbrace{\begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-2} \\ f_{n-1} \end{pmatrix}}_{\boldsymbol{f}}, \tag{8.4}$$

or, more succinctly as

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{g}$$

where $\boldsymbol{g} = h^2 \boldsymbol{f}$.

It is common to represent the finite difference formula as a stencil, see Figure 8.4. By sweeping this across the grid points on our mesh, it will generate the linear equations given in (8.4).

We now make some remarks regarding the properties of $\boldsymbol{A}$: it is a sparse matrix; more precisely, it is a tridiagonal matrix. We also note that $\boldsymbol{A}$ is symmetric (i.e., $\boldsymbol{A} = \boldsymbol{A}^\mathsf{T}$) and positive definite (i.e., $\boldsymbol{v}^\mathsf{T}\boldsymbol{A}\boldsymbol{v} > 0$ for all vectors $\boldsymbol{v} \in \mathbb{R}^{n-1}$, $\boldsymbol{v} \neq \boldsymbol{0}$).

The system of $n-1$ equations is solvable and has a unique solution

$$\boldsymbol{u} = \begin{pmatrix} u_1 & u_2 & \cdots & u_{n-1} \end{pmatrix}^\mathsf{T}.$$

The error at the grid points is of second order, i.e. $|u(x_i) - u_i| \sim O(h^2)$.

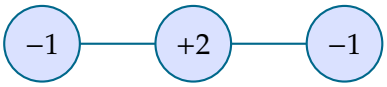


Figure 8.4: Weights in the three-point finite difference stencil for the approximation of $-h^2 u_{xx}$.

Nonhomogeneous Dirichlet boundary conditions

If $u_0, u_n \neq 0$, we can write the $n-1$ equations (8.1) as

$$\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & & \ddots & & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{pmatrix} = h^2 \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-2} \\ f_{n-1} \end{pmatrix} + \underbrace{\begin{pmatrix} u_0 \\ 0 \\ \vdots \\ 0 \\ u_n \end{pmatrix}}_{\boldsymbol{b}}. \tag{8.5}$$

This system can again be expressed on the form

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{g},$$

where the left hand side is the same as before. However, the right hand side is now $\boldsymbol{g} = h^2 \boldsymbol{f} + \boldsymbol{b}$, where the additional vector $\boldsymbol{b}$ is defined in (8.5).

8.4 Two-dimensional Poisson problem

We now consider the Poisson problem in a rectangular domain with lengths L_x and L_y; see Figure 8.5. The Poisson problem we consider can be expressed as

$$\begin{aligned} -\nabla^2 u &= f \qquad \text{in } \Omega, \\ u &= 0 \qquad \text{on } \partial\Omega. \end{aligned}$$

where the right hand side $f(x, y)$ (the source term) is assumed to be known.

Finite difference discretization

The finite difference grid points (or nodes) in Figure 8.6 are given by

$$\begin{aligned} x_i &= i \cdot h_x, \quad i = 0, 1, \ldots, m, \qquad h_x = \frac{L_x}{m}, \\ y_j &= j \cdot h_y, \quad j = 0, 1, \ldots, n, \qquad h_y = \frac{L_y}{n}. \end{aligned}$$

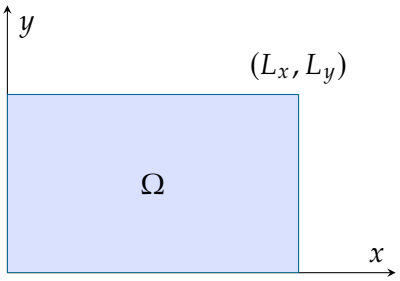


Figure 8.5: A rectangular domain for the two-dimensional Poisson problem.

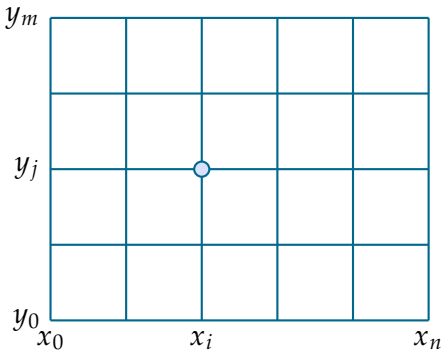


Figure 8.6: Finite difference grid: a structured grid.

Let $u_{i,j}$ be an approximation to $u(x_i, y_j)$, $1 \le i \le m-1$, $1 \le j \le n-1$, and let $f_{i,j} = f(x_i, y_j)$. Then

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} \simeq \left(\frac{\partial^2 u}{\partial x^2}\right)\Big|_{(x_i,y_j)} + O(h_x^2),$$
$$\frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h_y^2} \simeq \left(\frac{\partial^2 u}{\partial y^2}\right)\Big|_{(x_i,y_j)} + O(h_y^2).$$

Assuming (for simplicity) that $m = n$, and that $h_x = h_y = h$, the approximation of the Poisson problem at the internal grid points can be expressed as

$$-\frac{(u_{i+1,j} - 2u_{i,j} + u_{i-1,j})}{h^2} - \frac{(u_{i,j+1} - 2u_{i,j} + u_{i,j-1})}{h^2} = f_{i,j}, \qquad i, j = 1, \ldots, n-1,$$

or

$$-u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} + 4u_{i,j} = h^2 f_{i,j}, \qquad i, j = 1, \ldots, n-1. \quad (8.6)$$

We note that each unknown value $u_{i,j}$ is coupled to its nearest neighbors (north, south, east, west) according to the five-point stencil we are using; see Figure 8.7.

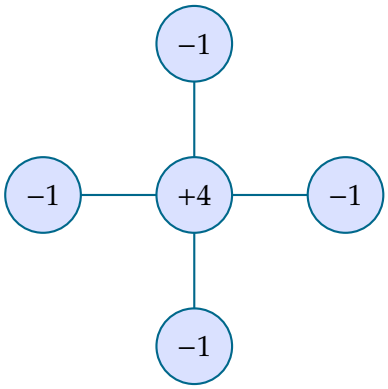


Figure 8.7: Weights in the five-point finite difference stencil for the approximation of $-h^2\nabla^2$.

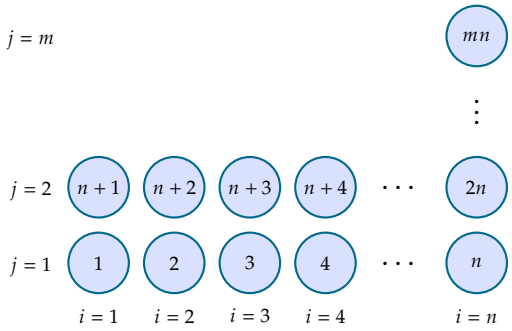


Figure 8.8: We use a "natural" ordering of the unknowns: we first number all the *internal* nodes along "row" 1 (in the x-direction), followed by the nodes in "row" 2, etc. The boundary nodes are not counted since these values are assumed to be known.

Global numbering scheme

In order to generate a matrix system as in the one-dimensional case, we need to define a unique ordering of all the degrees-of-freedom. We will here use a "natural" ordering in the sense that we number the *internal* nodes along the x-direction first, i.e.

$$u_{k=(n-1)(j-1)+i} \equiv u_{i,j}$$
$$f_{k=(n-1)(j-1)+i} \equiv f_{i,j}$$

Here, $k = 1, \ldots, N$ with $N = (n-1)^2$. We see that the ordering maps each node, (i, j), in the grid to a unique, global index, k; see Figure 8.8.

System of equations

With the chosen numbering scheme of the unknowns, we can express the discrete equations Equation 8.6 as

$$\underbrace{\begin{pmatrix} \boldsymbol{A}_0 & \boldsymbol{A}_1 & & & \\ \boldsymbol{A}_1 & \boldsymbol{A}_0 & \boldsymbol{A}_1 & & \\ & \boldsymbol{A}_1 & \boldsymbol{A}_0 & \ddots & \\ & & \ddots & \ddots & \boldsymbol{A}_1 \\ & & & \boldsymbol{A}_1 & \boldsymbol{A}_0 \end{pmatrix}}_{\boldsymbol{A}} \underbrace{\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_N \end{pmatrix}}_{\boldsymbol{u}} = \underbrace{\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_N \end{pmatrix}}_{\boldsymbol{g}} \tag{8.7}$$

where the matrices $\boldsymbol{A}_0$ and $\boldsymbol{A}_1$ are defined as

$$\boldsymbol{A}_0 = \begin{pmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & -1 & 4 & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 4 \end{pmatrix} \qquad \boldsymbol{A}_1 = \begin{pmatrix} -1 & & & & \\ & -1 & & & \\ & & -1 & & \\ & & & \ddots & \\ & & & & -1 \end{pmatrix}. \tag{8.8}$$

The dimension of this system is $N = (n-1)^2$. The matrix $\boldsymbol{A}$ is still sparse; in particular, it is pentadiagonal, reflecting the use of a five-point stencil for the approximation of the Laplace operator. However, while the bandwidth in the one-dimensional case is one, the bandwidth is now n.

Finally, we remark that the matrix $\boldsymbol{A}$ is symmetric and positive definite as in the one-dimensional case. This will guarantee that the system (8.7) is solvable and will yield a unique solution $\boldsymbol{u} = \begin{pmatrix} u_1 & \cdots & u_N \end{pmatrix}^\mathsf{T}$. The discretization error is still quadratic in the grid spacing h,

$$|u(x_i, y_j) - u_{i,j}| \sim O(h^2).$$

Solution methods for $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{g}$

Once we have generated the system of equations $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{g}$, we need to solve this system. There are two main classes of solution methods: direct and iterative methods. We now give a brief discussion of each of these classes.

Direct methods

A direct solution method is method which solves the system of algebraic equations in a finite, predictable number of steps, e.g., Gaussian elimination, FFT, etc. This class of the solution methods is often robust, especially when $\boldsymbol{A}$ is symmetric and positive definite.

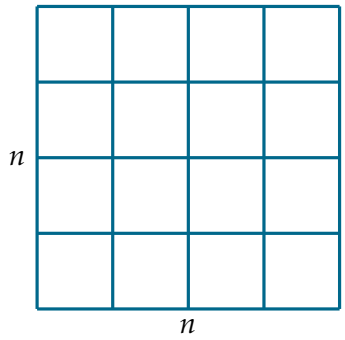


Figure 8.9: The two-dimensional grid associated with the finite difference solution of the Poisson problem. Here, $N \sim n^2$, while the bandwidth of $\boldsymbol{A}$ using a natural numbering scheme is $b \sim n$.

In the following, we will use the following notation:

$$\begin{aligned} \mathcal{N}_{\text{op}} &= \text{number of floating-point operations} \\ \mathcal{M} &= \text{memory requirement (in number of bytes)} \end{aligned}$$

We will also assume that $\boldsymbol{A} \in \mathbb{R}^{N \times N}$.

As an example, consider full LU-factorization (Gaussian elimination). In this case, we do not exploit any information about sparsity of $\boldsymbol{A}$, but treat this as a full matrix. For this solution approach, we have

$$\begin{aligned} \mathcal{N}_{\text{op}} &\sim O(N^3), \\ \mathcal{M} &\sim O(N^2). \end{aligned}$$

Full LU-factorization is robust and easy to use, however the cost does not scale very well since we need $O(N^2)$ operations per unknown. For large systems, this cost becomes prohibitive.

A better approach is to exploit the banded structure of $\boldsymbol{A}$. For a banded LU-factorization of bandwidth b, we have

$$\begin{aligned} \mathcal{N}_{\text{op}} &\sim O(Nb^2), \\ \mathcal{M} &\sim O(Nb). \end{aligned}$$

We remark that, since $\boldsymbol{A}$ is symmetric and positive definite, no pivoting is necessary during the Gaussian elimination process.

Let us now revisit (8.7) which we arrived at based on a finite difference discretization of the two-dimensional Poisson problem. If we exploit the banded structure of $\boldsymbol{A}$ during the LU-factorization, see Figure 8.9, the cost of this method is

$$\begin{aligned} \mathcal{N}_{\text{op}} &\sim O(Nb^2) \sim O(n^4) \sim O(N^2) \\ \mathcal{M} &\sim O(Nb) \sim O(n^3) \sim O(N^{3/2}) \end{aligned}$$

However, if we instead use FFT (The Fast Fourier Transform), as we will discuss later, the cost of solving (8.7) is only

$$\mathcal{N}_{op} \sim O(N \log N),$$
$$\mathcal{M} \sim O(N).$$

This is approximately optimal since the computational cost is $O(\log N)$ per unknown, and the memory requirement is constant per unknown, independent of the value of N.

Iterative methods

An iterative method will give a new estimate or update of the solution at each iteration. In most cases, the exact solution to $\boldsymbol{Au} = \boldsymbol{g}$ will never be reached. However, one can get as close as one wishes, but this may require many iterations and imply a high computational cost. Unlike a direct method, the solution is not reached after a finite, predictable number of floating point operations. In order to stop the iteration, a stop criterion has to be specified by the user. This can be some kind of tolerance, e.g. that the residual (i.e. the difference between the left hand side and the right hand side of the equation system) is less than a tolerance. Typically, when the tolerance is reduced, the number of iterations increases.

Another aspect with iterative methods is that the computational cost can be problem dependent; see Figure 8.10. This is different from direct solution methods which often only depends on the problem size, N.

A great advantage of many iterative methods is that they often have a *scalable* memory requirement, i.e. that the memory requirement is proportional to N and not some power of N.

Another characteristics with iterative methods is that much of the computational cost is related to performing basic linear algebra operations such as matrix-vector products, inner products, etc.

Finally, we remark that iterative methods are very suitable for parallel processing, and are often the only viable alternative for large, three-dimensional problems.

Some examples of iterations methods suitable for symmetric and positive definite problems are: Jacobi iteration, Gauss-Seidel iteration, the conjugate gradient method, and multigrid methods (where the last two methods are commonly used today).

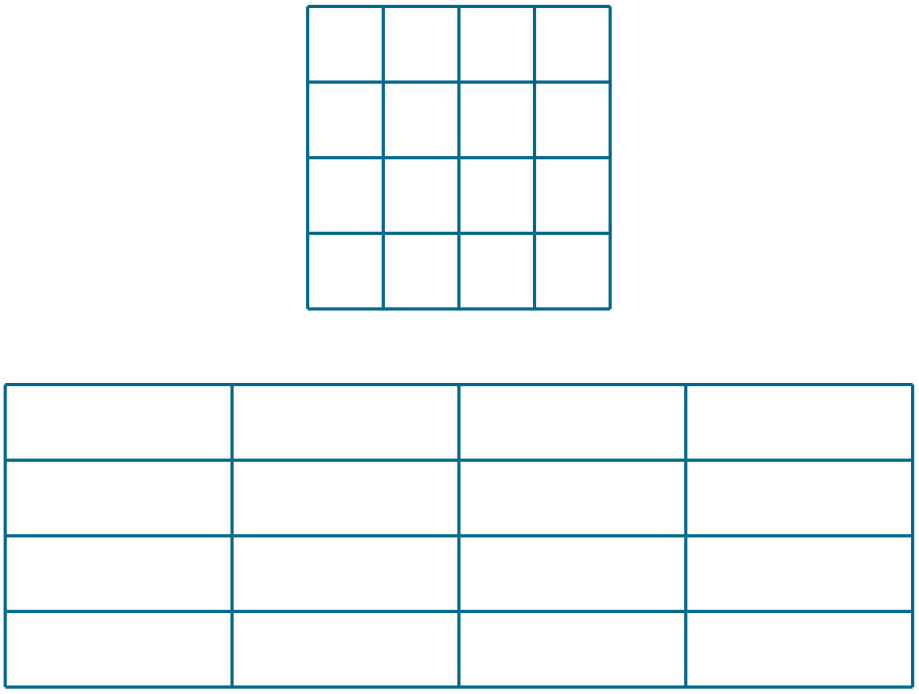


Figure 8.10: Two different computational domains for the Poisson problem, but with the same number of unknowns after discretization. The problem size, N, is therefore the same. The number of iterations required to solve the system of algebraic equations will depend on the aspect ratio (the ratio L_x/L_y) of the domains. Hence, the computational cost for solving the problem on the right will be higher than for the problem on the left. This is in contrast to a direct method where the computational cost for solving the two systems will be the same.

Chapter 9

Diagonalization for Poisson

9.1 Direct method based on diagonalization

Let us now consider a different way of solving the finite difference equations we derived in the context of discretizing the Poisson problem. The method is based on *diagonalization*, and we first explain the approach in the context of the one-dimensional Poisson problem:

$$\begin{aligned} -u_{xx} &= f \quad \text{in}\,\Omega = (0,1), \\ u(0) = u(1) &= 0. \end{aligned}$$

Assume that we use a uniform finite difference grid given by:

$$x_i = x_0 + ih, \quad i = 0, 1, \ldots, n.$$

The corresponding system of algebraic equations can be written as

$$\frac{1}{h^2}\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & & \ddots & -1 \\ & & & -1 & 2 \end{pmatrix}\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-1} \end{pmatrix},$$

where u_i is an approximation to $u(x_i) = u(ih)$, $i = 1, \ldots, n-1$, $f_i = f(x_i)$, and $u_0 = u_n = 0$ due to the specified boundary conditions. Let us write this system as

$$\frac{1}{h^2}\boldsymbol{T}\boldsymbol{u} = \boldsymbol{f}$$

where

$$\boldsymbol{T} = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & & \ddots & -1 \\ & & & -1 & 2 \end{pmatrix}, \qquad \boldsymbol{u} = \begin{pmatrix} u_1 \\ \vdots \\ u_{n-1} \end{pmatrix}, \qquad \boldsymbol{f} = \begin{pmatrix} f_1 \\ \vdots \\ f_{n-1} \end{pmatrix},$$

and h is the grid size or mesh size. Since $\boldsymbol{T}$ is symmetric positive definite, it can be *diagonalized*.

Diagonalization of $\boldsymbol{T}$

Diagonalization of $\boldsymbol{T}$ means that we wish to find the eigenvalues λ_j and the eigenvectors $\boldsymbol{q}_j$ of $\boldsymbol{T}$,

$$\boldsymbol{T}\boldsymbol{q}_j = \lambda_j \boldsymbol{q}_j \quad j = 1, \ldots, n-1,$$

where

$$\begin{aligned} \lambda_j > 0 \qquad & \text{(positive eigenvalues)}, \\ \boldsymbol{q}_k^{\mathsf{T}} \boldsymbol{q}_j = \delta_{jk} \qquad & \text{(orthonormal eigenvectors)}. \end{aligned}$$

We collect all the eigenvectors $\boldsymbol{q}_j$ into the orthogonal matrix $\boldsymbol{Q}$,

$$\boldsymbol{Q} = \begin{pmatrix} \boldsymbol{q}_1 & \boldsymbol{q}_2 & \cdots & \boldsymbol{q}_{n-1} \end{pmatrix}.$$

Then

$$\boldsymbol{T}\boldsymbol{Q} = \boldsymbol{Q}\boldsymbol{\Lambda}$$

where

$$\boldsymbol{\Lambda} = \operatorname{diag}(\lambda_1, \ldots, \lambda_{n-1}) = \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_{n-1} \end{pmatrix}.$$

Since

$$\boldsymbol{Q}^{\mathsf{T}}\boldsymbol{Q} = \boldsymbol{I} = \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix} \qquad \Rightarrow \quad \boldsymbol{Q}^{\mathsf{T}} = \boldsymbol{Q}^{-1},$$

and

$$\boldsymbol{T} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{\mathsf{T}} \tag{9.1}$$

or

$$\boldsymbol{Q}^\top \boldsymbol{T} \boldsymbol{Q} = \boldsymbol{\Lambda} \qquad \text{(diagonal)}.$$

Following this approach the finite difference approximation can be computed as follows:

$$\boldsymbol{g} \equiv h^2 \boldsymbol{f} \qquad \boldsymbol{g}: \ O(n) \text{ operations}$$

$$\begin{aligned} \boldsymbol{T}\,\boldsymbol{u} &= \boldsymbol{g} \\ \boldsymbol{Q}\,\boldsymbol{\Lambda}\,\boldsymbol{Q}^\top \boldsymbol{u} &= \boldsymbol{g} \\ \boldsymbol{\Lambda}\,\underbrace{\boldsymbol{Q}^\top \boldsymbol{u}}_{\tilde{\boldsymbol{u}}} &= \underbrace{\boldsymbol{Q}^\top \boldsymbol{g}}_{\tilde{\boldsymbol{g}}} \qquad \tilde{\boldsymbol{g}}: \ O(n^2) \text{ operations} \end{aligned}$$

$$\begin{aligned} \boldsymbol{\Lambda}\tilde{\boldsymbol{u}} &= \tilde{\boldsymbol{g}} \\ \tilde{\boldsymbol{u}} &= \boldsymbol{\Lambda}^{-1}\tilde{\boldsymbol{g}} \qquad \tilde{\boldsymbol{u}}: \ O(n) \text{ operations} \end{aligned}$$

$$\begin{aligned} \boldsymbol{Q}^\top \boldsymbol{u} &= \tilde{\boldsymbol{u}} \\ \boldsymbol{u} &= \boldsymbol{Q}\tilde{\boldsymbol{u}} \qquad \boldsymbol{u}: \ O(n^2) \text{ operations} \end{aligned}$$

Note that the transformations

$$\tilde{\boldsymbol{g}} = \boldsymbol{Q}^\top \boldsymbol{g} \quad \text{and} \quad \boldsymbol{u} = \boldsymbol{Q}\tilde{\boldsymbol{u}}$$

are *matrix-vector products*. In summary, we can compute $\boldsymbol{u}$ in

$$O(n) + O(n^2) + O(n) + O(n^2) \sim O(n^2) \text{ floating-point operations.}$$

Hence, we can solve our finite difference system in $(n-1)$ unknowns in $O(n^2)$ operations. This is *not* competitive with a direct solution algorithm based upon LU-factorization (Gaussian elimination) of a tridiagonal matrix, which can be done in $O(n)$ operations (since the bandwidth is equal to one).

Let us also compare the memory requirement:

$O(n^2)$ for the diagonalization approach (we need to store $\boldsymbol{Q}$);
$O(n)$ for a tridiagonal direct solver.

Again, the diagonalization approach is *not* competitive.

So, why bother? The answer is that the diagonalization approach becomes more interesting in $\mathbb{R}^2$. In addition, we will later see how we can use the Fast Fourier Transform (FFT) to lower the computational complexity.

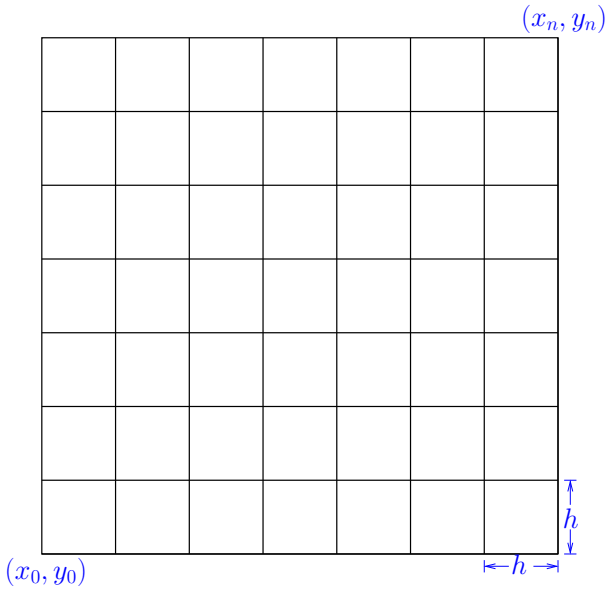


Figure 9.1: A uniform finite difference grid.

9.2 The Poisson problem in $\mathbb{R}^2$

The two-dimensional Poisson problem on the unit square is given by

$$\begin{aligned} -\nabla^2 u &= f \qquad \text{in } \Omega = (0,1) \times (0,1), \\ u &= 0 \qquad \text{on } \partial\Omega, \end{aligned} \tag{9.2}$$

where

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

Again, using the notation $u_{i,j} \simeq u(x_i, y_j) = u(ih, jh)$ and $f_{i,j} = f(x_i, y_j)$, and discretizing (9.2) using the 5-point stencil (see Figure 9.1), the discrete equations read

$$-\frac{(u_{i+1,j} - 2u_{i,j} + u_{i-1,j})}{h^2} - \frac{(u_{i,j+1} - 2u_{i,j} + u_{i,j-1})}{h^2} = f_{i,j} \tag{9.3}$$

for $1 \le i, j \le n-1$.

Diagonalization

Let

$$\boldsymbol{U} = \begin{pmatrix} u_{1,1} & \cdots & u_{1,n-1} \\ \vdots & \ddots & \vdots \\ u_{n-1,1} & \cdots & u_{n-1,n-1} \end{pmatrix}$$

and

$$\boldsymbol{T} = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}.$$

Then,

$$\begin{aligned} (\boldsymbol{TU})_{ij} &= 2u_{i,j} - u_{i+1,j}, && i = 1, \\ (\boldsymbol{TU})_{ij} &= -u_{i-1,j} + 2u_{i,j} - u_{i+1,j}, && 2 \le i \le n-2, \\ (\boldsymbol{TU})_{ij} &= -u_{i-1,j} + 2u_{i,j}, && i = n-1. \end{aligned}$$

and thus,

$$\frac{1}{h^2}(\boldsymbol{TU})_{ij} \simeq -\left(\frac{\partial^2 u}{\partial x^2}\right)_{i,j}. \tag{9.4}$$

Similarly in the other direction,

$$\frac{1}{h^2}(\boldsymbol{UT})_{ij} \simeq -\left(\frac{\partial^2 u}{\partial y^2}\right)_{i,j}. \tag{9.5}$$

Our finite difference system (9.3) can thus be expressed as

$$\frac{1}{h^2}(\boldsymbol{TU} + \boldsymbol{UT})_{ij} = f_{i,j} \qquad \text{for} \qquad 1 \le i, j \le n-1,$$

or

$$\boldsymbol{TU} + \boldsymbol{UT} = \boldsymbol{G} \tag{9.6}$$

where

$$\boldsymbol{G} = h^2 \begin{pmatrix} f_{1,1} & \cdots & f_{1,n-1} \\ \vdots & \ddots & \vdots \\ f_{n-1,1} & \cdots & f_{n-1,n-1} \end{pmatrix}.$$

Combining (9.1) and (9.6) we get

$$\boldsymbol{Q\Lambda Q^\intercal U} + \boldsymbol{UQ\Lambda Q^\intercal} = \boldsymbol{G}. \tag{9.7}$$

Multiplying (9.7) from the right with $\boldsymbol{Q}$ and from the left with $\boldsymbol{Q}^\intercal$, and using the fact that $\boldsymbol{Q}^\intercal\boldsymbol{Q} = \boldsymbol{I}$, we get

$$\boldsymbol{\Lambda}\underbrace{\boldsymbol{Q^\intercal U Q}}_{\tilde{\boldsymbol{U}}} + \underbrace{\boldsymbol{Q^\intercal U Q}}_{\tilde{\boldsymbol{U}}}\boldsymbol{\Lambda} = \underbrace{\boldsymbol{Q^\intercal G Q}}_{\tilde{\boldsymbol{G}}}.$$

Hence, (9.6) may be solved in three steps:

1. Compute $\tilde{\boldsymbol{G}}$ using matrix-matrix products

$$\tilde{\boldsymbol{G}} = \boldsymbol{Q}^\intercal \boldsymbol{G} \boldsymbol{Q}.$$

2. Solve for $\tilde{\boldsymbol{U}}$.

$$\begin{aligned} \boldsymbol{\Lambda}\tilde{\boldsymbol{U}} + \tilde{\boldsymbol{U}}\boldsymbol{\Lambda} &= \tilde{\boldsymbol{G}} \\ \lambda_i \tilde{u}_{ij} + \tilde{u}_{ij}\lambda_j &= \tilde{g}_{ij} \\ \tilde{u}_{ij} &= \frac{\tilde{g}_{ij}}{\lambda_i + \lambda_j} \end{aligned}$$

3. Compute $\boldsymbol{U}$ using matrix-matrix products

$$\boldsymbol{U} = \boldsymbol{Q}\tilde{\boldsymbol{U}}\boldsymbol{Q}^\intercal$$

Here,

$$\boldsymbol{U}, \tilde{\boldsymbol{U}}, \tilde{\boldsymbol{G}}, \boldsymbol{Q} \in \mathbb{R}^{(n-1)\times(n-1)}$$

Computational cost

The number of degrees of freedom (or unknowns) N is

$$N = (n-1)^2 \sim O(n^2) \quad (n \gg 1).$$

1. Two matrix-matrix products give $O(n^3)$ operations.
2. $O(n^2)$ scalar constant time operations give $O(n^2)$.
3. Two matrix-matrix products give $O(n^3)$ operations.

In summary, we can compute the discrete solution, $\boldsymbol{U}$, in $O(n^3) = O(N^{3/2})$ operations.

Note: this method is an example of a *direct method*.

Comparison with other direct methods

We conclude that the diagonalization method is much more attractive in $\mathbb{R}^2$ than in $\mathbb{R}^1$. The number of floating-point operations per degree of freedom is $O(n)$, while the memory requirement is close to optimal (i.e. scalable).

Table 9.2: Computational cost and memory requirement for three different direct methods. The number of unknowns is $N = O(n^2)$. For the banded solver, we have used a bandwidth $b \sim O(n)$. Full LU means LU-factorization without exploiting sparsity.

Method	Operations ($\mathcal{N}_{\text{op}}$)	Memory requirement ($\mathcal{M}$)
Diagonalization	$O(N^{3/2}) = O(n^3)$	$O(N) = O(n^2)$
Banded LU	$O(Nb^2) = O(n^4)$	$O(Nb) = O(n^3)$
Full LU	$O(N^3) = O(n^6)$	$O(N^2) = O(n^4)$

The matrices Q and Λ

The computational cost associated with the diagonalization approach tacitly assumes that we know the eigenvector matrix $\boldsymbol{Q}$ and the corresponding eigenvalues. Let us therefore derive explicit expressions for these. To this end, consider first the continuous eigenvalue problem

$$\begin{aligned} -u_{xx} &= \lambda u \qquad \text{in } \Omega = (0,1), \\ u(0) = u(1) &= 0, \end{aligned}$$

with solutions

$$\begin{aligned} \bar{u}_j(x) &= \sin(j\pi x), \\ \bar{\lambda}_j &= j^2\pi^2, \end{aligned} \qquad j = 1, 2, \ldots, \infty.$$

Consider now the discrete eigenvalue problem

$$\boldsymbol{T}\tilde{\boldsymbol{q}}_j = \lambda_j \tilde{\boldsymbol{q}}_j$$

Try eigenvector solutions which correspond to the continuous eigenfunctions $\bar{u}_j(x)$ sampled at the grid points x_i, $i = 1, \ldots, n-1$, i.e.

$$\begin{aligned} (\tilde{\boldsymbol{q}}_j)_i &= \bar{u}_j(x_i) \\ &= \sin(j\pi x_i) \\ &= \sin(j\pi(ih)), \\ &= \sin\left(\frac{ij\pi}{n}\right) \end{aligned}$$

Operating on $\tilde{\boldsymbol{q}}_j$ with $\boldsymbol{T}$ gives

$$(\boldsymbol{T}\tilde{\boldsymbol{q}}_j)_i = \underbrace{2\left(1 - \cos\left(\frac{j\pi}{n}\right)\right)}_{\lambda_j} \underbrace{\sin\left(\frac{ij\pi}{n}\right)}_{(\tilde{\boldsymbol{q}}_j)_i}$$

Hence, our try was successful: operating on $\tilde{\boldsymbol{q}}_j$ with $\boldsymbol{T}$ gives a multiple of $\tilde{\boldsymbol{q}}_j$.

In order to proceed, set $\boldsymbol{q}_j = \alpha \tilde{\boldsymbol{q}}_j$, and choose α such that $\boldsymbol{q}_j$ is normalized:

$$(\boldsymbol{q}_j)_i = \sqrt{\frac{2}{n}} \sin\left(\frac{ij\pi}{n}\right), \qquad 1 \le i, j \le n-1,$$
$$\lambda_j = 2\left(1 - \cos\left(\frac{j\pi}{n}\right)\right).$$

For $j \ll n$, we observe that

$$\lambda_j \simeq 2\left(1 - \left(1 - \frac{1}{2}\frac{j^2\pi^2}{n^2} + \ldots\right)\right) \simeq \frac{j^2\pi^2}{n^2}.$$

Since $h = 1/n$, we have

$$\lambda_j \simeq h^2 j^2 \pi^2 = h^2 \bar{\lambda}_j \qquad \text{for } j \ll n.$$

Since the approximation of the one-dimensional Laplace operator on our finite difference grid is equal to $\boldsymbol{T}/h^2$, this is the same as saying that the first, lowest eigenvalues (and eigenvectors) for the continuous case are well approximated by our finite difference formulation.

Note that

$$\boldsymbol{Q}_{ij} = (\boldsymbol{q}_j)_i = \sqrt{\frac{2}{n}} \sin\left(\frac{ij\pi}{n}\right), \qquad 1 \le i, j \le n-1,$$

and that indeed

$$\boldsymbol{Q}^\mathsf{T} = \boldsymbol{Q}, \qquad \boldsymbol{Q}^\mathsf{T}\boldsymbol{Q} = \boldsymbol{I}.$$

From the comparison of the computational cost shown earlier, the diagonalization approach to solving the discrete Poisson problem appears promising.

Questions:

1. Can the matrix-matrix multiplications be done fast?
2. Can the matrix-matrix multiplications be parallelized?
3. Can we do better?

Table 9.3: Simulation results for the numerical approximation of the two-dimensional Poisson equation by finite differences. The solution of the system of discrete equations is based on diagonalization techniques and matrix-matrix products. A listing of the FORTRAN program used in these tests is given below. It is interesting to note that the elapsed time on a single processor on Gridur is reduced by a factor of more than 80 compared to using the PC from 2000 when $n = 1024$.

	PC (2000)			Gridur (2006)	
n	$\tau(n)$	$\tau_1(n)$	$r(n)$	$\tau(n)$	$\tau_1(n)$
32	$1.80 \cdot 10^{-2}$	$1.76 \cdot 10^{-5}$			
64	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-5}$	2.1		
128	1.20	$7.34 \cdot 10^{-5}$	2.0	$2.45 \cdot 10^{-2}$	$1.50 \cdot 10^{-6}$
256	9.84	$1.50 \cdot 10^{-4}$	2.0	0.167	$2.55 \cdot 10^{-6}$
512	103.9	$3.96 \cdot 10^{-4}$	2.6	1.33	$5.08 \cdot 10^{-6}$
1024	873.2	$8.33 \cdot 10^{-4}$	2.1	10.33	$9.85 \cdot 10^{-6}$
	$\sim O(n^3)$	$\sim O(n)$			

Numerical results

A diagonalization solver based on "standard" matrix-matrix product code (i.e. no special library was used). The code was run on a PC, Pentium III with 512 MB RAM (2000) and on a single processor on Gridur, MIPS R14000 with 1 GM RAM (2006).

We define $\tau(n)$ as the total simulation time (in seconds), $\tau_1(n) = \tau(n)/n^2$ as the time spent per degree of freedom, and

$$r(n) = \frac{\tau_1(n)}{\tau_1(n/2)}$$

as the slowdown factor when doubling the problem size.

See Table 9.3. The source code follows.

```
program poisson

! Program to solve the two-dimensional Poisson equation on
! a unit square using one-dimensional eigenvalue decompositions
! and matrix-vector products.
! In this example, the right hand side f=1.

! Einar M. Ronquist
! NTNU, October 2000
```

```
parameter (n = 256)
parameter (m = n-1)

real*8 diag(m), q(m,m), qt(m,m), b(m,m), u(m,m), w(m,m), pi
real*4 tarray(2), t1, t2, dt

t1 = etime(tarray)

h  = 1./n
pi = 4.*atan(1.)

do i=1,m
  diag(i) = 2*(1-cos(i*pi/n))
enddo

do j=1,m
  do i=1,m
    q(i,j) = sin(i*j*pi/n) * sqrt((2./n))
  enddo
enddo

do j=1,m
  do i=1,m
    qt(i,j) = q(j,i)
  enddo
enddo

do j=1,m
  do i=1,m
    b(i,j) = h*h
  enddo
enddo

call mxm(b,m,q,m,w,m)
call mxm(qt,m,w,m,b,m)

do j=1,m
  do i=1,m
    u(i,j) = b(i,j)/(diag(i)+diag(j))
  enddo
enddo

call mxm(u,m,qt,m,w,m)
```

```
call mxm(q,m,w,m,u,m)

t2 = etime(tarray)
dt = t2-t1
write(6,*) ' '
write(6,*) 'dt (total)= ',dt
dt = dt/(n*n)
write(6,*) 'dt (per dof)= ',dt

stop
end

subroutine mxm (a,n1,b,n2,c,n3)

! matrix-matrix product c = a*b

real*8 a(n1,n2), b(n2,n3), c(n1,n3)
do j=1,n3
  do i=1,n1
    c(i,j) = 0.0
    do k=1,n2
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    enddo
  enddo
enddo

return
end
```

Fast diagonalization methods

The most expensive operation in the diagonalization method introduced in the previous section is of the type

$$v^* = Qv = Q^T v,$$

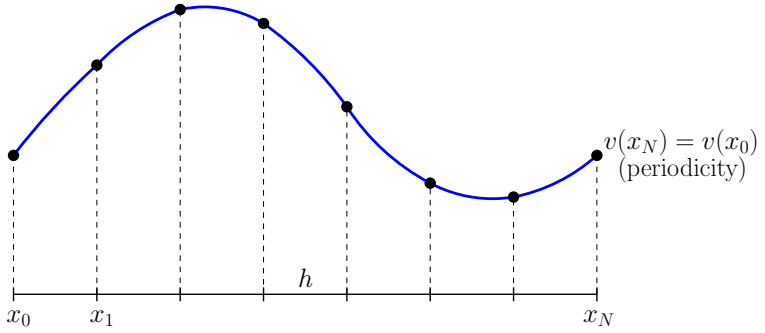
where

$$Q_{ij} = \sqrt{\frac{2}{n}} \sin\left(\frac{ij\pi}{n}\right) \qquad 1 \le i, j \le n-1.$$

We will now consider ways to obtain v^* in $O(n \log n)$ operations instead of $O(n^2)$.

Discrete Fourier Transform (DFT)

Consider a periodic function $v(x)$ with period 2π. Consider sampling this function at the equidistant points x_j, $j = 0, 1, \ldots, N$ with $x_j = jh, \quad h = 2\pi/N$. Let $v_j = v(x_j) = v(jh)$, $j = 0, 1, \ldots, N$.



Consider the vectors $\boldsymbol{\varphi}_k$, where

$$(\boldsymbol{\varphi}_k)_j = \mathrm{e}^{\mathrm{i}kx_j}, \qquad j, k = 0, 1, \ldots, N-1.$$

Note that the vector elements in $\boldsymbol{\varphi}_k$ represent the values of the function $\varphi_k(x) = \mathrm{e}^{\mathrm{i}kx}$ sampled at the discrete points $x_j, \quad j = 0, 1, \ldots, N-1$. Note also that the function $\varphi_k(x) = \mathrm{e}^{\mathrm{i}kx}$ is an eigenfunction of the Laplace operator with periodic boundary conditions.

The vectors $\{\boldsymbol{\varphi}_k\}_{k=0}^{N-1}$ form a basis for the N-dimensional vector space $\mathbb{C}^N$. In particular, we have that

$$\boldsymbol{\varphi}_k^{\mathsf{H}} \boldsymbol{\varphi}_l = N\delta_{kl}, \qquad k, l = 0, 1, \ldots, N-1.$$

The vector

$$\boldsymbol{v} = \begin{pmatrix} v_0 & \cdots & v_{N-1} \end{pmatrix}^{\mathsf{T}} \in \mathbb{R}^N$$

can be expressed in this basis as

$$\boldsymbol{v} = \sum_{k=0}^{N-1} \hat{v}_k \boldsymbol{\varphi}_k \qquad \Rightarrow \qquad v_j = \sum_{k=0}^{N-1} \hat{v}_k (\boldsymbol{\varphi}_k)_j = \sum_{k=0}^{N-1} \hat{v}_k \mathrm{e}^{\mathrm{i}kx_j},$$

where $\hat{v}_k$, are the discrete Fourier coefficients given by

$$\hat{v}_k = \frac{1}{N} \sum_{j=0}^{N-1} v_j \mathrm{e}^{-\mathrm{i}kx_j}, \qquad \begin{matrix} x_j = jh \\ h = 2\pi/N \end{matrix} \qquad k = 0, 1, \ldots, N-1$$

Discrete Sine Transform (DST)

The Discrete Sine Transform is applicable to a function $v(x)$ which is *periodic* with period 2π and *odd*. Discretize the function on an equidistant grid on

$[0, \pi]$ with $h = \pi/n$. Set

$$v_j = v(x_j) = v(jh) = v\left(\frac{j\pi}{n}\right), \qquad j = 0, 1, \ldots, n.$$

Since v is odd,

$$v(x_0) = v(x_n) = 0.$$

The discretized function is therefore represented by the $(n-1)$ real values $v_1, \ldots, v_{n-1}$, i.e. by the vector

$$\boldsymbol{v} = \begin{pmatrix} v_1 & \vdots & v_{n-1} \end{pmatrix}^\mathsf{T} \in \mathbb{R}^{n-1}.$$

An orthogonal basis for $\mathbb{R}^{n-1}$ is given by the vectors $\boldsymbol{\psi}_k$, $k = 1, \ldots, n-1$, where

$$(\boldsymbol{\psi}_k)_j = \sin\left(\frac{kj\pi}{n}\right), \qquad j = 1, \ldots, n-1,$$

and with

$$\boldsymbol{\psi}_k^\mathsf{T} \boldsymbol{\psi}_l = \begin{cases} \frac{n}{2}, & k = l, \\ 0, & k \neq 0. \end{cases}$$

In terms of this basis, we can write $\boldsymbol{v}$ as

$$v_j = \sum_{k=1}^{n-1} \tilde{v}_k \sin\left(\frac{kj\pi}{n}\right), \qquad j = 1, \ldots, n-1,$$

where

$$2\tilde{v}_k = \frac{2}{n} \sum_{j=1}^{n-1} v_j \sin\left(\frac{jk\pi}{n}\right), \qquad k = 1, \ldots, n-1.$$

We can also write this as $\tilde{\boldsymbol{v}} = \boldsymbol{S}\boldsymbol{v}$ (DST) and $\boldsymbol{v} = \boldsymbol{S}^{-1}\tilde{\boldsymbol{v}}$ (inverse DST). Note that $\boldsymbol{S}$ and $\boldsymbol{S}^{-1}$ are related as

$$\boldsymbol{S} = \frac{2}{n}\boldsymbol{S}^{-1}$$

Also note that

$$\boldsymbol{Q} = \sqrt{\frac{2}{n}}\boldsymbol{S}^{-1} = \sqrt{\frac{n}{2}}\boldsymbol{S}.$$

Now, consider the matrix $\boldsymbol{F}^{(N)}$ where

$$\begin{aligned} F_{k,j}^{(N)} &= \mathrm{e}^{-\mathrm{i}jkh} \\ &= \cos\left(\frac{jk2\pi}{N}\right) - i\sin\left(\frac{jk2\pi}{N}\right), \qquad 0 \leq j, k \leq N-1. \end{aligned}$$

Note that

$$F^{(2n)}_{k,j} = \cos\left(\frac{jk\pi}{n}\right) - i\sin\left(\frac{jk\pi}{n}\right), \qquad 0 \le j,k \le 2n-1.$$

Now, consider

$$\boldsymbol{v} = \begin{pmatrix} v_1 & \cdots & v_{n-1} \end{pmatrix}^{\mathsf{T}} \in \mathbb{R}^{n-1}.$$

Construct the extended vector as an "odd" extension

$$\boldsymbol{w} = \begin{pmatrix} 0 & v_1 & \cdots & v_{n-1} & 0 & -v_{n-1} & \cdots & -v_1 \end{pmatrix}^{\mathsf{T}} \in \mathbb{R}^{2n}.$$

First, note that

$$(\boldsymbol{F}^{(2n)}\boldsymbol{w})_k = \sum_{j=0}^{2n-1} \mathrm{e}^{\frac{-ijk\pi}{n}} w_j = 2n\hat{w}_k,$$

where $\hat{w}$, $k = 0, 1, \ldots, 2n-1$ are the discrete Fourier coefficients. Second,

$$\begin{aligned}
(\boldsymbol{F}^{(2n)}\boldsymbol{w})_k &= \sum_{j=0}^{2n-1} \left[\cos\left(\frac{jk\pi}{n}\right) - i\sin\left(\frac{jk\pi}{n}\right)\right] w_j \\
&= \sum_{j=0}^{2n-1} w_j \cos\left(\frac{jk\pi}{n}\right) - i\sum_{j=0}^{2n-1} w_j \sin\left(\frac{jk\pi}{n}\right) \\
&= -2i\sum_{j=0}^{n-1} w_j \sin\left(\frac{jk\pi}{n}\right) \\
&= -2i\sum_{j=1}^{n-1} w_j \sin\left(\frac{jk\pi}{n}\right) \qquad (\text{since } w_0 = 0),
\end{aligned}$$

where the first sum vanishes because it is a product of an odd and an even sequence.

Hence

$$\frac{i}{2}(\boldsymbol{F}^{(2n)}\boldsymbol{w})_k = \sum_{j=1}^{n-1} w_j \sin\left(\frac{jk\pi}{n}\right) = \frac{n}{2}\tilde{w}_k.$$

Since

$$w_j = v_j, \qquad j = 1, \ldots, n-1,$$

it follows that

$$\widetilde{w}_k = \widetilde{v}_k, \qquad k = 1, \ldots, n-1.$$

In summary, for $k = 1, \ldots, n-1$,

$$\begin{aligned}\tilde{v}_k = \tilde{w}_k &= \frac{2}{n} \cdot \frac{\mathrm{i}}{2}(\boldsymbol{F}^{(2n)}\boldsymbol{w})_k \\ &= \frac{\mathrm{i}}{n}(\boldsymbol{F}^{(2n)}\boldsymbol{w})_k \\ &= \frac{\mathrm{i}}{n} 2n\hat{w}_k \\ &= 2\mathrm{i}\hat{w}_k.\end{aligned}$$

By computing the discrete Fourier coefficients $\hat{w}_k$, we can find the discrete sine coefficients $\widetilde{v}_k$, $k = 1, \ldots, n-1$, where

$$\tilde{\boldsymbol{v}} = \boldsymbol{S}\boldsymbol{v} = \sqrt{\frac{2}{n}}\boldsymbol{Q}\boldsymbol{v}.$$

The operator $(\boldsymbol{F}^{(2n)}\boldsymbol{w})$ can be computed efficiently by a FFT in $O(2n \log 2n) \sim O(n \log n)$ operations.

This leads to the *modified algorithm* for Poisson:

1. Compute $\tilde{\boldsymbol{G}}^\intercal$ in $O(n^2 \log n)$.

$$\begin{aligned}\tilde{\boldsymbol{G}} &= \boldsymbol{Q}^\intercal \boldsymbol{G} \boldsymbol{Q} \\ \Rightarrow \quad \tilde{\boldsymbol{G}}^\intercal &= \boldsymbol{Q}^\intercal \boldsymbol{G}^\intercal \boldsymbol{Q} \\ &= \boldsymbol{Q}\boldsymbol{G}^\intercal\boldsymbol{Q}^\intercal \qquad (\boldsymbol{Q} = \boldsymbol{Q}^\intercal) \\ &= \boldsymbol{Q}(\boldsymbol{Q}\boldsymbol{G})^\intercal \\ &= \sqrt{\frac{2}{n}}\boldsymbol{S}^{-1}\sqrt{\frac{n}{2}}(\boldsymbol{S}\boldsymbol{G})^\intercal \\ &= \boldsymbol{S}^{-1}(\boldsymbol{S}\boldsymbol{G})^\intercal\end{aligned}$$

2. Compute $\tilde{\boldsymbol{U}}^\intercal$ in $O(n^2)$.

$$\tilde{U}_{ji} = \frac{\tilde{G}_{ji}}{\lambda_j + \lambda_i}$$

3. Compute $\boldsymbol{U}$ in $O(n^2 \log n)$.

$$\begin{aligned}\boldsymbol{U} &= \boldsymbol{Q}\tilde{\boldsymbol{U}}\boldsymbol{Q}^\intercal \\ &= \boldsymbol{Q}(\boldsymbol{Q}\tilde{\boldsymbol{U}}^\intercal)^\intercal \\ &= \boldsymbol{S}^{-1}(\boldsymbol{S}\tilde{\boldsymbol{U}}^\intercal)^\intercal\end{aligned}$$

Again, recall that

$$\boldsymbol{S} = \frac{2}{n}\boldsymbol{S}^{-1}$$

and $\tilde{\boldsymbol{v}} = \boldsymbol{S}\boldsymbol{v}$ is obtained as

1. $\boldsymbol{v} \in \mathbb{R}^{n-1} \quad \rightarrow \quad \boldsymbol{w} \in \mathbb{R}^{2n}$
2. Compute $\hat{\boldsymbol{w}}$ via FFT in $O(n \log n)$.
3. $\tilde{v}_k = 2\mathrm{i}\hat{w}_k, \qquad k = 1, \ldots, n-1.$

Numerical results

A diagonalization solver based on the FFT. The code was run on a PC, Pentium III with 512 MB RAM.

We define $\tau(n)$ as the total simulation time (in seconds) and $\tau_1(n) = \tau(n)/n^2$ as the time spent per degree of freedom.

Table 9.4: Simulation results for the numerical approximation of the two-dimensional Poisson equation by the use of fast diagonalization techniques.

n	$\tau(n)$	$\tau_1(n)$	$\tau_1(n)(m \times m)$
32	$2.36 \cdot 10^{-2}$	$2.31 \cdot 10^{-5}$	$1.76 \cdot 10^{-5}$
64	$1.11 \cdot 10^{-1}$	$2.71 \cdot 10^{-5}$	$3.66 \cdot 10^{-5}$
128	$5.19 \cdot 10^{-1}$	$3.17 \cdot 10^{-5}$	$7.34 \cdot 10^{-5}$
256	2.35	$3.58 \cdot 10^{-5}$	$1.50 \cdot 10^{-4}$
512	10.5	$3.99 \cdot 10^{-5}$	$3.96 \cdot 10^{-4}$
1024	46.2	$4.41 \cdot 10^{-5}$	$8.33 \cdot 10^{-4}$
	$\sim O(n^2 \log n)$	$\sim O(\log n)$	$\sim O(n)$

See Table 9.4. The source code follows.

```
program poisson

! FORTRAN-program to solve the two-dimensional Poisson equation
! on a unit square using finite differences (five-point stencil),
! one-dimensional eigenvalue decompositions and fast sine transforms
! In this example, the right hand side f=1.

! note: n needs to be a power of 2

! Einar M. Ronquist
! NTNU, October 2000

parameter (n  = 256)
parameter (m  = n-1)
parameter (nn = 4*n)
```

```
real*8    diag(m), b(m,m), bt(m,m)
real*8    pi
real*8    z(0:nn-1)
real*4    tarray(2), t1, t2, dt

h    = 1./n
pi   = 4.*datan(1.)

do i=1,m
  diag(i) = 2*(1-dcos(i*pi/n))
enddo

do j=1,m
  do i=1,m
    b(i,j) = h*h
  enddo
enddo

do j=1,m
  call fst (b(1,j), n, z, nn)
enddo
call transp (bt, b, m)
do i=1,m
  call fstinv (bt(1,i), n, z, nn)
enddo

do j=1,m
  do i=1,m
    bt(i,j) = bt(i,j)/(diag(i)+diag(j))
  enddo
enddo

do i=1,m
  call fst (bt(1,i), n, z, nn)
enddo
call transp (b, bt, m)
do j=1,m
  call fstinv (b(1,j), n, z, nn)
enddo

umax = 0.0
do j=1,m
  do i=1,m
```

```
      if (b(i,j) .gt. umax) umax = b(i,j)
    enddo
  enddo

  write(6,*) ' '
  write(6,*) umax

  stop
  end

  subroutine transp (at, a, m)

  ! set at equal to the transpose of a

  real*8 a(m,m), at(m,m)

  do j=1,m
    do i=1,m
      at(j,i) = a(i,j)
    enddo
  enddo

  return
  end
```

Chapter 10

Parallel I/O in MPI

10.1 Introduction

As HPC applications grow larger and larger as more and more computing resources are made available, so does the volume of data which needs to be handled, both on the input and the output side of the application. The process of reading data into a process or storing data from a process we hereby refer to as *I/O*. The input side is mostly a solved problem, in the sense that most operating systems and filesystems added support for multiple processes reading from the same file years ago.

Getting high performance, however, is more involved. For the last thirty years or so, secondary storage has usually been hard disk drives (commonly abbrevated as *HDDs*); see Figure 10.1 for an illustration of the interior. These are mechanical in nature, data is read/written from/to spinning platters through a magnetic device referred to as the *head*. Since a HDD only has a single head, it can only perform a single read/write operation concurrently. This means that even though multiple processes can initiate reads concurrently, they will be performed in serial. Schematically this can be illustrated as in Figure 10.2. A HDD only performs at peak levels if data is read or written in large sequential chunks, since searching for data incur large penalties, as the head has to be repositioned. In a lot of HPC applications, the access pattern for a single process can seem to be fairly random from the operating system's point of view, which lead to the I/O being performed in an inefficient manner. This is due to the fact that the data a particular process needs is scattered in the file, leading to excessive seeks. Only when all the I/O performed by the separate processes are considered as a whole, a sequential pattern can be found. The application developer is often aware of this fact, and this raises the need for an interface where this information can be given to the system. Additional performance can be achieved if a file is stored across multiple storage devices. This allows us to harness the aggregated bandwith of the devices; see Figure 10.3 for an illustration.

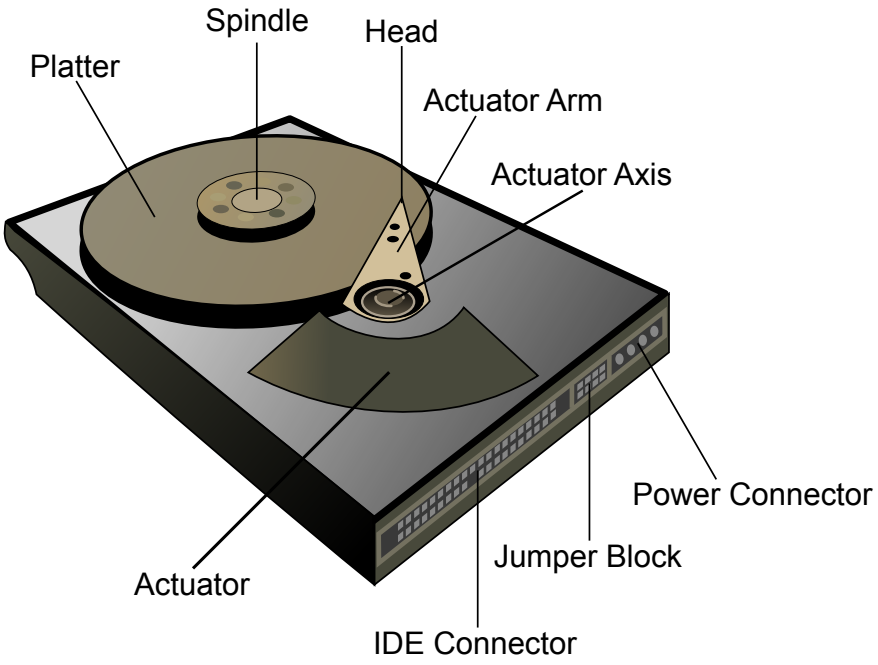


Figure 10.1: Illustration of the interior of a hard disk drive. The data is stored on platters. The data is stored magnetically on these platters by a magnetic device referred to as the *head*. The same device is responsible for retrieving the data when a read operation is issued. Since we only have a single head, HDDs are serial by construction. Image taken from http://en.wikipedia.org/wiki/Harddrive.

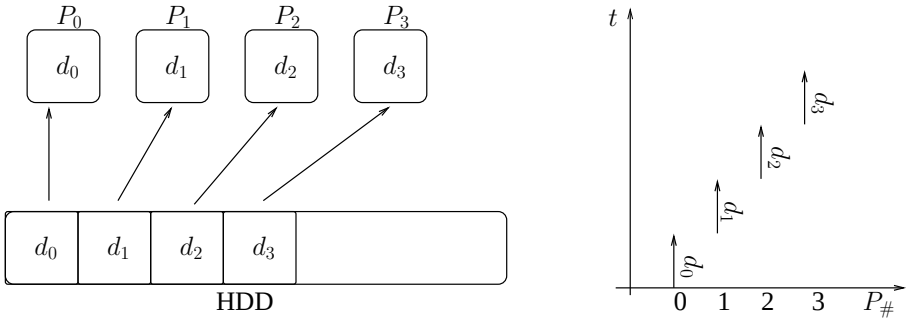


Figure 10.2: Illustration of I/O where several processes read from a single file. Since a HDD is serial in nature, only one operation can be performed concurrently.

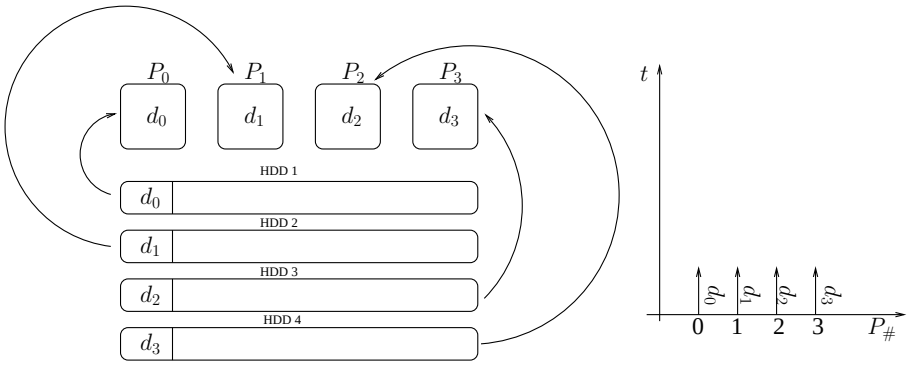


Figure 10.3: Illustration of I/O where we harness the aggregated bandwith of several devices. Here each process reads its data from a separate physical device, thus the I/O can be performed in parallel.

On the output side, things turn are interesting. For years HPC applications solved their needs using one of two approaches. The first approach is often referred to as *post-mortem assembly*; each process dumps its data to a separate file, and then custom-tailored code is used to read the data into the next application in the computational chain, such as visualization software. If implementing the support directly into the visualization software is not feasible, a separate postprocessing application must be written. The second approach, which is also often used, is to serialize the I/O. Here one process is given the responsibility of writing the data to disc. The other processes then simply send their data to the responsible process, in a sequential manner; see Figure 10.4.

What is common to both of these approaches, is that they necessitate performing (most of) the I/O in a serial manner. As the data volume grows, this occupies an increasing amount of the total computational time. Eventually this may become a large bottleneck for the parallel performance of your program. The first approach also necessitates performing substantially more I/O than strictly required, in particular all data is written twice; first to the separate files and then in a stitched-together fashion as performed by the postprocessing application. If the volume of data is substantial, this introduces another severe bottleneck in your computational chain. The code involved in both of these approaches is often intricate and prone to errors.

It is thus of great interest to be able to store all data in a single file, or at most a few files, while avoiding serialization of the I/O both on the application level, as well as on the physical level, by splitting the file across multiple storage devices. Fortunately there are established libraries to enable this. We here consider one particular example of such an interface, namely the *MPI-IO* interface. As the name implies, this was made part of the official

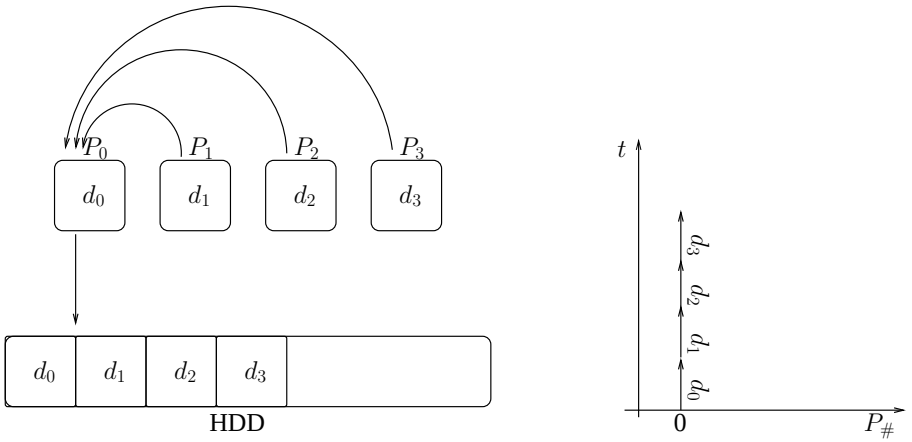


Figure 10.4: Illustration of serialization of the I/O. All processes send their data to process 0, which is responsible for writing the data to secondary storage. Since a harddrive is serial in nature, the total time spent on the operation equals the sum of the time to write the individual data chunks.

MPI standard back in 1998 with the introduction of the MPI 2.0 specification. However, while the library strive to yield highly portable code, tuning to the underlying filesystem (how the data is stored on the hard disk drive(s)) is unavoidable when it comes to obtaining high performance. One example of such a file system is the *general parallel filesystem*, GPFS for short, developed by IBM during the last 10 years. This is in use on NOTUR systems (*Kongull*, *Vilje*). Since such tuning is somewhat technical and beyond the intention of this document, this will not be discussed in detail in the following.

10.2 Basic concepts of MPI-IO

MPI-IO is the part of the official MPI standard which covers parallel I/O. It offers a simple, natural interface for expressing parallelism in I/O. Informing the system about the underlying distributed nature of an I/O call enables the system to make good choices about how the I/O is performed on a lower level. We first consider the simplest case: I/O that is sequential on each process.

Sequential I/O

We here consider sequential I/O, namely writing a contiguous slice of data from a set of separate processes. The data storage pattern in this case is fairly trivial, as illustrated in Figure 10.5. An example where such a pattern could occur, is if we have partitioned a matrix in a strip fashion across the processes; see Figure 10.7a for an illustration.

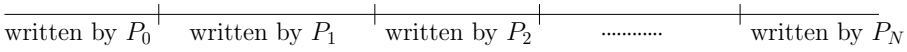


Figure 10.5: Illustration of an I/O operation where a contiguous piece of data is sliced into chunks. A single contiguous chunk of data is then written by each process. All the chunks are written to the same file.

If you are familiar with standard POSIX I/O calls, the basic form of the MPI-IO code should look very familiar. On each process we start by opening a handle to the file we want to write to, in this case called "datafile".

```
MPI_File fh;
MPI_File_open(MPI_COMM_WORLD, "datafile",
              MPI_MODE_WRONLY|MPI_MODE_CREATE,
              MPI_INFO_NULL, &fh);
```

A file handle is described by a variable of type `MPI_File`. In addition to which file we want to open, we also have to specify flags which the system use to both control which accesses are allowed, as well as for directives which help increase performance. Here we only specify flags of the former class, in particular, we only want to write to the file (by specifying the flag `MPI_MODE_WRONLY`) and we want the file to be created if it doesn't already exist (by specifying the flag `MPI_MODE_CREATE`). The function includes an additional parameter of type `MPI_Info`. For now we pass the builtin value `MPI_INFO_NULL` which can be used when we do not want to pass any data in this field. We will briefly comment on this later.

With the file opened, we proceed with writing the data. In this example, our data is an array of doubles named `vec`. The total global array length is `N`, which means we have `N/size` elements per process (here `size`, as usual, denotes the number of processes, while `rank` denotes the process number). We first seek to the appropriate offset in the file, then write the chunk of data.

```
MPI_Offset mysize = N/size;

MPI_File_seek(fh, rank*mysize*sizeof(double),
              MPI_SEEK_SET);
MPI_File_write(fh, vec, mysize, MPI_DOUBLE,
               MPI_STATUS_IGNORE);
```

An alternative way of doing the same operation, is to use the *explicit offset* `MPI_File_Write_at` function. This is mainly a matter of convenience, i.e. it allows to do the same operation with less code, in particular, we avoid the seek call;

```
MPI_File_write_at(fh, rank*mysize*sizeof(double),
                  vec, mysize, MPI_DOUBLE,
                  MPI_STATUS_IGNORE);
```

Common to both of these alternatives is that they use individual I/O calls on each process. If the underlying system is smart, it should be able to recognize the access pattern. If we want to make sure that the system notices this fact, we can use *collective* calls. In particular, consider

```
MPI_File_seek(fh, rank*mysize*sizeof(double),
              MPI_SEEK_SET);
MPI_File_write_all(fh, vec, mysize,
                   MPI_DOUBLE, MPI_STATUS_IGNORE);
```

Here we inform the system that we are performing a collective write among all the processes. This allows the system to do additional coordination of how the I/O is performed. However, it comes at a cost. In particular, all processes will stall until the entire I/O operation is completed across all the processes, while using individual handles allows the program flow to continue on each process as soon as their share of the I/O has been performed.

Another convenience function in the context of collective calls is *shared* writes. In addition to containing info about the position on the individual processes, a `MPI_File` variable also contains info about a shared state between all the processes. Here, with our simple data layout, this can be exploited to make the code particularly simple. We simply have to do

```
MPI_File_write_ordered(fh, vec, mysize,
                       MPI_DOUBLE, MPI_STATUS_IGNORE);
```

This function uses the shared state in the file pointer. Upon the call, the writes are performed in order according to process rank.

Finally we have to close our file. This is performed by doing

```
MPI_File_close(&fh);
```

10.3 Non-sequential I/O: file views

In the previous section we considered an extremely simple data access pattern. Such a pattern is only the predominant one in simple codes. Usually the access patterns in real applications are much more involved. Programming such access patterns naively using many seeks and many small writes is certainly a possibility, however hardly something we would recommend. In addition to being highly error prone, it would also make it very hard for the system to get a proper view of the collective I/O performed across the processes.

For purposes of easy presentation, we here again consider a simple data layout. In particular, consider a vector split in a cyclic manner, see Figure 10.6.

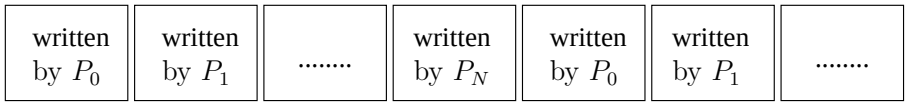


Figure 10.6: Illustration of cyclic partition of a vector. Each process writes a single number, followed by a gap of size − 1 numbers. This pattern repeats for the extent of the data.

If we perform this I/O naively, each process would write a single number, perform a seek, write another number and so on. Such small writes followed by seeks makes it very hard for the system to optimize the I/O. MPI-IO offers a much better approach. It builds on the MPI machinery for constructing custom types. While these datatypes are used to describe the data layout in memory in standard MPI functions, they are here used to describe the data layout on secondary storage. From each process, the data access pattern is one number followed by a gap of size − 1 numbers. We can describe such a pattern in a MPI datatype as

```
MPI_Datatype filetype;
MPI_Type_create_resized(MPI_DOUBLE, 0,
                        size*sizeof(double),
                        &filetype);
MPI_Type_commit(&filetype);
```

We here use the function

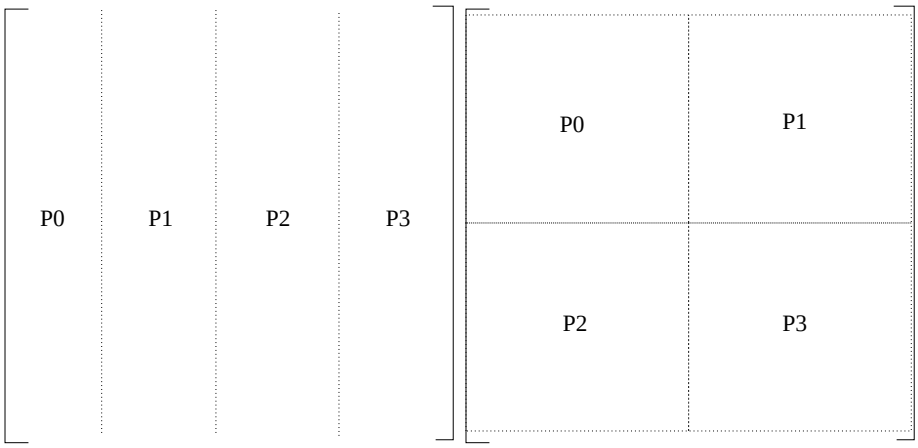
```
MPI_Type_create_resized(origtype, lb, extent, newtype)
```

to create the gap. We set `lb` (lower bound) = 0 to indicate that we want the real data to start at the beginning of the new data type. We then extend this to be size doubles long. We can now attach this view of the data to the file using the function

```
MPI_File_set_view(fh, disp, datatype,
                  viewtype, encoding, info)
```

as

```
MPI_File_set_view(fh, rank*sizeof(double),
                  MPI_DOUBLE, filetype, "native",
                  MPI_INFO_NULL);
```



(a) Strips. (b) Blocks.

Figure 10.7: Illustration of two possible ways a 2D array can be distributed across several processes. In Figure 10.7a each process is responsible for a number of whole columns. In Figure 10.7b each process is instead reponsible for a sub-block of the array.

Note the usage of the *disp* field. This field describes an offset that the process sees as the start of the file. By setting this offset equal to position of the first number the particular process should write to the file, the data access pattern (*MPI_Datatype*) we described further up is exactly the same on each process. With this configured, writing the data to the file in the proper pattern is now just a normal write call, e.g. using separate handles

```
MPI_File_write(f,vec,mysize,
               MPI_DOUBLE,MPI_STATUS_IGNORE);
```

A code demonstrating this can be found in Appendix B.1.

We stress that there is nothing stopping you from using different fileviews on the different processes. In fact, we now move on to consider a particular case where this is needed, namely when we want to handle arrays/matrices partitioned over several processes.

10.4 Non-sequential I/O: distributed arrays

In HPC applications, your data usually consists of matrices (2D arrays) or 3D arrays. This fact has not gone past the MPI creators, and as such MPI contains machinery for generating partioning of such arrays in a semi-automatic way. Since MPI-IO is built on MPI, this machinery is also extremely useful when we want to write the data to secondary storage. We now show how to employ these utility functions to make writing distributed arrays to disc as simple as performing a single write call.

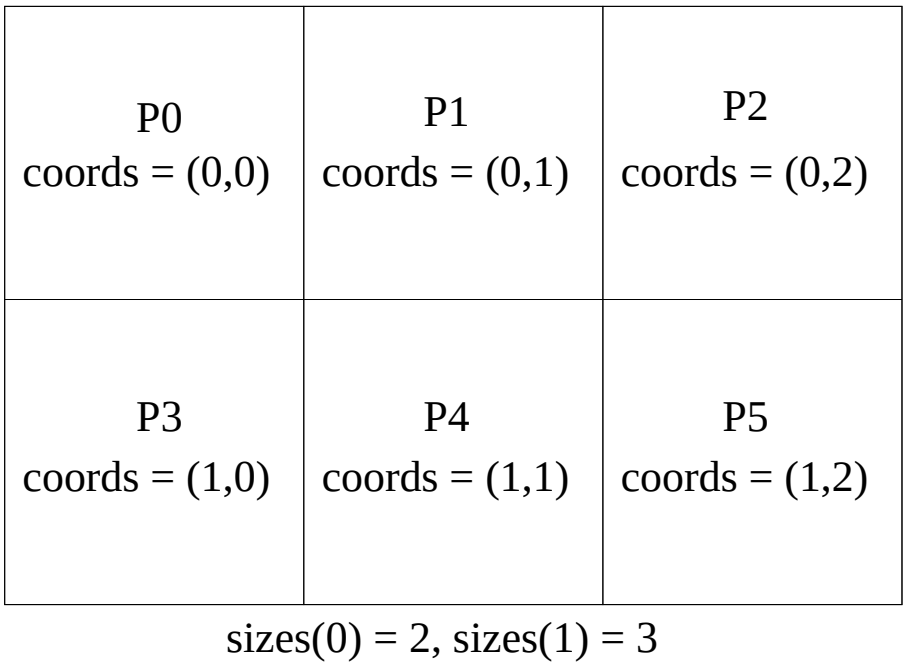


Figure 10.8: A block partition of a 2D array with the pieces of information we need to classify this partitioning.

MPI names such distributed arrays *darray* for short. The available functions are heavily inspired by the High Performance Fortran standard, HPF for short. This is a standard which aims to make developing HPC software easier through semi-automatic parallelization. In this context, the most important parts of this standard is its conventions for array partitioning strategies and process topologies, since MPI follows the same conventions. The two possible partitionings of most interest in this context is illustrated in Figure 10.7. As we will show shortly, these are to some extent similar.

We now show how to use the MPI machinery to easily partition an array. Consider Figure 10.8. We here have a 2D array we want to partition across 6 MPI processes. The figure contains the pieces of information we need to classify a partitioning:

- A global topology: here a Cartesian topology expressed as the number of processes along each dimension (here 2 and 3, respectively).
- Location of a particular domain in the topology, again this can be expressed as an integer along each dimension.
- A mapping of the available processes onto the topology.

The first useful function is `MPI_Dims_create`. This function generates a Cartesian partioning of your processes according to a the rules set by HPF.

```
int sizes[2];
sizes[0] = sizes[1] = 0;
MPI_Dims_create(size, 2, sizes);
```

The initialization of the entries in sizes prior to calling the function is important. In particular, the function will only operate along a dimension i where `sizes[i] = 0` on function entry. While this interface makes it very easy for the programmer to make errors (i.e. forgetting to initialize the array properly), it also has advantages. For instance, if we instead want to divide our matrix in a strip fashion, we can do

```
int sizes[2];
sizes[0] = 1; = sizes[1] = 0;
MPI_Dims_create(size, 2, sizes);
```

Of course, if we only have one partitioned dimension in our array, generating the topology is trivial. The nice thing with doing it like this is that it allows us to use fairly similar code, whether we want a strip or a block partitioning of the arrays; see Figure 10.7. We stress that this only generates the *topology* (or partitioning structure), it does not in any way depend on the array dimensions. Upon return from the function the `dims` array contains the number of processes used in the separate directions, 2 and 3, respectively, in our example.

Now we have a topology describing the layout of our processors, or rather how many processors the dimensions are split over. Still, each processor needs to know where they are located in this topology. To be able to decide this, we first have to create a communicator group which has the (Cartesian) topology attached. This can be achieved by

```
int periodic[2]; periodic[0] = periodic[1] = 0;
MPI_Comm comm;
MPI_Cart_create(MPI_COMM_WORLD, 2,
                dims,periodic, 0, &comm);
```

The `periodic` array is an integer array with either 0 or 1 as entries. These are used to specify whether or not the domain is periodic in the particular dimension, which is not the case with a standard 2D array as the one we consider here. Upon return from the function, the `comm` variable holds the new communicator info. This can now be used wherever MPI expects a communicator (i.e. instead of the builtin `MPI_COMM_WORLD` we have used previously). This takes care of mapping the processes onto the topology.

Finally, each process can then find their location in the topology using

```
int coords[2];
MPI_Cart_coords(comm,rank,2,coords);
```

Upon return from the function, the `coords` array holds the coordinates. In our example, it would for instance hold 1 and 2 when called on process 5.

With the problem partitioning taken care of, it is time to tie this into MPI-IO. MPI includes functions to describe such a distributed array layout in memory, which usually are useful when you want to collect a whole array on a single process. Collecting all data on a single process is exactly what we want to perform when we write this to secondary storage, the only difference is that in our case this "process" is the file on secondary storage. Thus we can use the functions originally intended for describing the data layout in memory to instead generate the fileviews on the separate processes. The function we need is

```
MPI_Type_create_darray(size, rank, dims, gsizes,
                       distribs, dargs, sizes,
                       order, etype, newtype)
```

Here `size` is the size of the communicator (typically the number of processes), `rank` the process rank within the communicator, `dims` the number of dimensions in the array (2 for a matrix), `gsizes` the sizes of the global array along the dimensions, `distribs` distribution strategies along the dimensions, `dargs` a distribution strategy parameter (can usually be set to `MPI_DISTRIBUTE_DFLT_DARG`), `dims` the topology results as obtained using `MPI_Dims_create`, `order` describes the array layout in memory (`MPI_ORDER_C` or `MPI_ORDER_FORTRAN`), `etype` the datatype of the array entries and finally `newtype` our new datatype.

The most interesting of these are the `distribs` array. This array contains the chosen distribution strategy along each dimension. The strategy can be one of

- `MPI_DISTRIBUTE_NONE`: Here no partitioning of the array is applied along this dimension.
- `MPI_DISTRIBUTE_CYCLIC`: Here a cyclic partitioning of the array is applied along this dimension. This distribution is often used in HPF codes. We do not consider it here.
- `MPI_DISTRIBUTE_BLOCK`: Here a block partitioning of the array is applied along this dimension.

For instance, if we want to block-partition an array of doubles with size $N \times N$ stored according to C convections (row-major storage), we do

```
int gsizes[2], distribs[2], dargs[2];
gsizes[0] = gsizes[1] = N;
distribs[0] = MPI_DISTRIBUTE_BLOCK;
distribs[1] = MPI_DISTRIBUTE_BLOCK;
dargs[0] = dargs[1] = MPI_DISTRIBUTE_DFLT_DARG;
```

```
MPI_Type_create_darray(size, rank, 2, gsizes,
                       distribs, dargs, sizes,
                       MPI_ORDER_C,
                       MPI_DOUBLE, filetype);
MPI_Type_commit(&filetype);
```

If we now set this datatype as the fileview on the individual processes as described earlier, we can now write them to disc using a single write call. This is much much easier than the alternative using separate write/seek calls.

A code that handles both strip and block partitioned arrays can be found in Appendix B.2.

10.5 Overlapping I/O and computations

On modern architectures HDDs can write/read data (almost) completely on their own using a technique known as *Direct Memory Access*, *DMA* for short. This means that while we are writing/reading data the CPU is mostly an idle observer. This is not good for program efficiency, ideally we would like to keep the CPUs saturated with work whenever we can. MPI-IO also offers facilities to remedy this, namely *non-blocking* I/O. All classes of I/O calls have non-blocking equivalents, to simplify the presentation we here only consider using individual handles on each process.

The basic idea can be summarized in the following piece of (somewhat) abstract code;

```
MPI_Request req;
MPI_File_iwrite(f, vec, mysize, MPI_DOUBLE, &req);
doSomething();
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

We have here replaced the `MPI_File_write` function with a call to `MPI_File_iwrite`, which is the nonblocking equivalent. This call initiates the I/O operation, then immediately returns. The function takes an additional parameter of type `MPI_Request`. Upon the return from the function call, this will be updated with information which identifies the I/O operation. We are now free to perform additional calculations, as long as the data we just requested written to secondary storage is not touched by this code. In particular, the vector `vec` cannot be updated in the `doSomething()` call. When we get to a point where we need write access to the vector `vec` again, we do a call to `MPI_Wait` with the `MPI_Request` variable as a parameter. This function will only return once it is safe to reuse `vec`. Note that a call to a nonblocking I/O function *ALWAYS* needs to be accompanied with a call to `MPI_Wait`, even if you do not plan to reuse the memory area you requested written to secondary storage.

10.6 Tuning for performance

As mentioned in the introduction, a program using MPI-IO needs to be tuned to the particular underlying filesystem if we want the best performance. This tuning can be divided in two classes.

- **Directives**: This class of tuning parameters are something an implementation has to obey. One example of such a tuning parameter is the flag `MPI_MODE_SEQUENTIAL` which can be passed upon opening the file, just like we passed e.g. `MPI_MODE_WRONLY` earlier. This tells the system that only sequential access to the data is performed, information which can be used to optimze the I/O.
- **Hints**: These are, as the name indicates, just hints to the implementation. If an implementation does not support/utilize a particular hint, they can just be silently ignored. This is the framework in which vendor specific/filesystem specific tuning can be performed.

These hints are passed to the implementation in a variable of type `MPI_Info`. First we need to create the appropriate structure, this is achieved through

```
MPI_Info info;
MPI_Info_create(&info);
```

We can now set hints in this structure. A hint is a pair of (key,value) strings. We add such a hint to our `MPI_Info` variable using the function

```
MPI_Info_set(info, key, value)
```

Such a call might look like

```
MPI_Info_set(&info,"access_style","write_mostly");
```

Once we have added all hints we want to the `MPI_Info` variable, we can now pass these hints to the implementation upon opening a file.

```
MPI_File fh;
MPI_File_open(MPI_COMM_WORLD,"datafile",
              MPI_MODE_WRONLY|MPI_MODE_CREATE,
              info,&fh);
```

We here pass our `MPI_Info` variable instead of `MPI_INFO_NULL` as earlier.

10.7 Further reading

You can find the MPI-IO standard as well as tutorials on the official MPI homepage http://www.mpi-forum.org/. Some useful papers and slides can be found at https://computing.llnl.gov/?set=code&page=sio_papers_presentations. These

are of particular interest if you want extensive knowledge about using MPI-IO on top of GPFS.

Acknowledgements: Jørn Amundsen gave some useful pointers while this chapter was written, in particular the link to llnl.gov. Tobias Arrskog helped with proof reading. The chapter was written by Arne Morten Kvarving. Your assistance was greatly appreciated.

Bibliography

[1] How to use OpenMP. https://computing.llnl.gov/tutorials/openMP/.

[2] The OpenMP specification. http://openmp.org.

[3] David E Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[4] Craig C Douglas, Gundolf Haase, Jonathan Hu, Markus Kowarschik, Ulrich Rüde, and Christian Weiss. Portable memory hierarchy techniques for pde solvers: Part ii. *Siam News*, 33(6):1, 2000.

[5] J. P. Prost, R. Treumann, R. Blackmore, C. Hartman, R. Hedges, B. Jia, A. Koniges, and A. White. Towards a High-Performance Robust Implementation of MPI-IO on top of GPFS. Preprint, available at https://computing.llnl.gov/code/sio/ucrl-jc-137128.pdf, 2000.

[6] B Lande. *Optimaliserting av flyttallsintensive programmer*. Institutt for matematiske fag, NTNU, 2004.

[7] (none given). Parallel I/O Techniques. Workshop slides, available at http://www.osc.edu/supercomputing/training/pario/parallel-io-nov04.pdf, 2003.

[8] Rajeev Thakur. Parallel I/O in MPI. Lecture slides, available at http://www.classes.cs.uchicago.edu/archive/2000/fall/CS103-01/Lectures/mpi-io/, 2000.

Appendix A

OpenMP code

A.1 Calculating π with an integral in OpenMP

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include <omp.h>

typedef double(*function_t)(double x);

double integrate(double x0, double x1, int n, function_t f)
{
  double h = (x1-x0)/n;
  double result=0.0;
#pragma omp parallel for schedule(static) reduction(+:result)
  for (int i=0;i<n;++i) {
    double x = x0 + (i+0.5)*h;
    result += h*f(x);
  }

  return result;
}

double myf(double x)
{
  return 4.0/(1.0+x*x);
}

int main(int argc, char** argv)
{
```

```
  int n;
  double mypi;

  n = atoi(argv[1]);
  if (n <= 0) {
    printf("Error, %i intervals make no sense, bailing\n",n);
    exit(1);
  }

  double start = omp_get_wtime();
  mypi = integrate(0,1,n,myf);
  printf("elapsed: %f\n",omp_get_wtime()-start);

  printf("%1.16f\n",fabs(mypi-4.0*atan(1.0)));

  return 0;
}
```

Appendix B

MPI-IO code

B.1 Storage of a cyclic-partitioned vector using MPI-IO

```
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* This program partions a vector of a specified length
 * over the available MPI processes in a cyclic fashion.
 *       |p0|p1|..|pn|p0|p1|..|pn|
 * A specified amount of repetitions of this vector
 *  is then saved as "fileview.vec" using MPI-IO.
 */

void setupVector(double* V, int rank, int size, int N)
{
    int i;
    for( i=0;i<N;++i )
        V[i] = i*size+rank;
}

int main(int argc, char** argv)
{
    if( argc < 2 ) {
        printf("need atleast one parameter, N\n");
        exit(1);
    }
    int rank, size;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    int N = atoi(argv[1]);
    int perproc=N/size;
    int K;
    if( argc > 2 )
        K = atoi(argv[2]);
    else
        K = 1;

    double* vec = (double*)malloc(perproc*sizeof(double));
    setupVector(vec,rank,size,perproc);

    MPI_Datatype filetype;
    MPI_Type_create_resized(MPI_DOUBLE,0,
                            size*sizeof(double),
                            &filetype);
    MPI_Type_commit(&filetype);

    MPI_File f;

    MPI_File_open(MPI_COMM_WORLD,"fileview.vec",
                  MPI_MODE_CREATE|MPI_MODE_RDWR,
                  MPI_INFO_NULL,&f);

    /* NOTE: THIS TRIGGERS A BUG IN OPENMPI v1.3 */
    MPI_File_set_view(f,rank*sizeof(double),MPI_DOUBLE,
                      filetype,"native",MPI_INFO_NULL);
    int n;
    for( n=0;n<K;++n )
        MPI_File_write(f,vec,perproc,
                       MPI_DOUBLE,MPI_STATUS_IGNORE);

    MPI_File_close(&f);

    MPI_Finalize();

    return 0;
}
```

B.2 Storage of a block/strip partitioned array using MPI-IO

```
#include <mpi.h>
#include <stdlib.h>
#include <memory.h>
#include <stdio.h>

/* This program partitions an array of a specified size
 * over the available MPI processes in either a strip
 * or a block fashion as specified.
 * A specified amount of repetitions of this array
 * is then saved as "combined.mat" using MPI-IO.
 */

void createFileView(MPI_Datatype* filetype, int* sizes,
                    int N, int rank, int size, int block)
{
    int gsizes[2], distribs[2], dargs[2];
    gsizes[0] = gsizes[1] = N;
    distribs[0] = MPI_DISTRIBUTE_BLOCK;
    sizes[0] = sizes[1] = 0;
    if( block ) {
        distribs[1] = MPI_DISTRIBUTE_BLOCK;
    }
    else {
        sizes[1] = 1;
        distribs[1] = MPI_DISTRIBUTE_NONE;
    }
    MPI_Dims_create(size,2,sizes);
    dargs[0] = dargs[1] = MPI_DISTRIBUTE_DFLT_DARG;
    MPI_Type_create_darray(size,rank,2,gsizes,distribs,
                           dargs,sizes,MPI_ORDER_C,
                           MPI_DOUBLE,filetype);
    MPI_Type_commit(filetype);
}

void setupMatrix(double** A, int rows, int cols,
                 int startrow, int startcol, int N)
{
    for( int i=0;i<rows;++i)
        for( int j=0;j<cols;++j )
            A[i][j] = (i+startrow)*N+j+startcol;
```

```
}

double** createMatrix(int n1, int n2)
{
    int i, n;
    double **a;
    a    = (double **)calloc(n1   ,sizeof(double *));
    a[0] = (double  *)calloc(n1*n2,sizeof(double));

    for (i=1; i < n1; i++)
        a[i] = a[i-1] + n2;

    return (a);
}

int main(int argc, char** argv)
{
    int rank, size;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    int N = atoi(argv[1]);
    int blocked = 0;
    if(argc > 2 )
        blocked = atoi(argv[2]);
    int sizes[2];
    MPI_Datatype filetype;
    createFileView(&filetype,sizes,N,rank,size,blocked);

    int periodic[2]; periodic[0] = periodic[1] = 0;
    MPI_Comm comm;
    MPI_Cart_create(MPI_COMM_WORLD,2,sizes,
                    periodic,0,&comm);
    int coord[2];
    MPI_Cart_coords(comm,rank,2,coord);

    int rows = N/sizes[0];
    int cols = N/sizes[1];
    int bufsize = rows*cols;
    int K;
    if( argc > 3 )
        K = atoi(argv[3]);
    else
```

```
        K = 1;

    double** A = createMatrix(rows,cols);
    setupMatrix(A,rows,cols,coord[0]*rows,coord[1]*cols,N);

    MPI_File f;
    MPI_File_open(MPI_COMM_WORLD,"combined.mat",
                  MPI_MODE_CREATE|MPI_MODE_RDWR,
                  MPI_INFO_NULL,&f);
    MPI_File_set_view(f,0,MPI_DOUBLE,
                      filetype,"native",MPI_INFO_NULL);
    for( int n=0;n<K;++n )
        MPI_File_write(f,A[0],bufsize,
                       MPI_DOUBLE,MPI_STATUS_IGNORE);

    MPI_File_close(&f);

    MPI_Finalize();

    return 0;
}
```