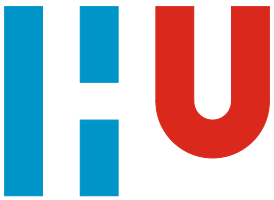
High Performance Processing

Huib Aldewereld



This reader was verified by:

Huib Aldewereld	hogeschoolhoofddocent
David Isaacs Paternostro	docent
Brian van der Bijl	docent

First release, January 2020
Second release, February 2021

Contents

Disclaimer

This reader was composed to support the course *High Performance Processing* of the HBO-ICT Bachelor programme, specialisation Artificial Intelligence, of the HU University of Applied Sciences. This reader is composed from large parts of the books Introduction to High Performance Scientific Computing [17] and Parallel Programming for Science and Engineering [16]. We are grateful that Victor Eijkhout developed and disclosed these materials under an open source license, and are glad to contribute to the effort.

1. Introduction

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

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

1.1 Scientific computing

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

As a definition, we may posit

> The efficient computation of constructive methods in applied mathematics.

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

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

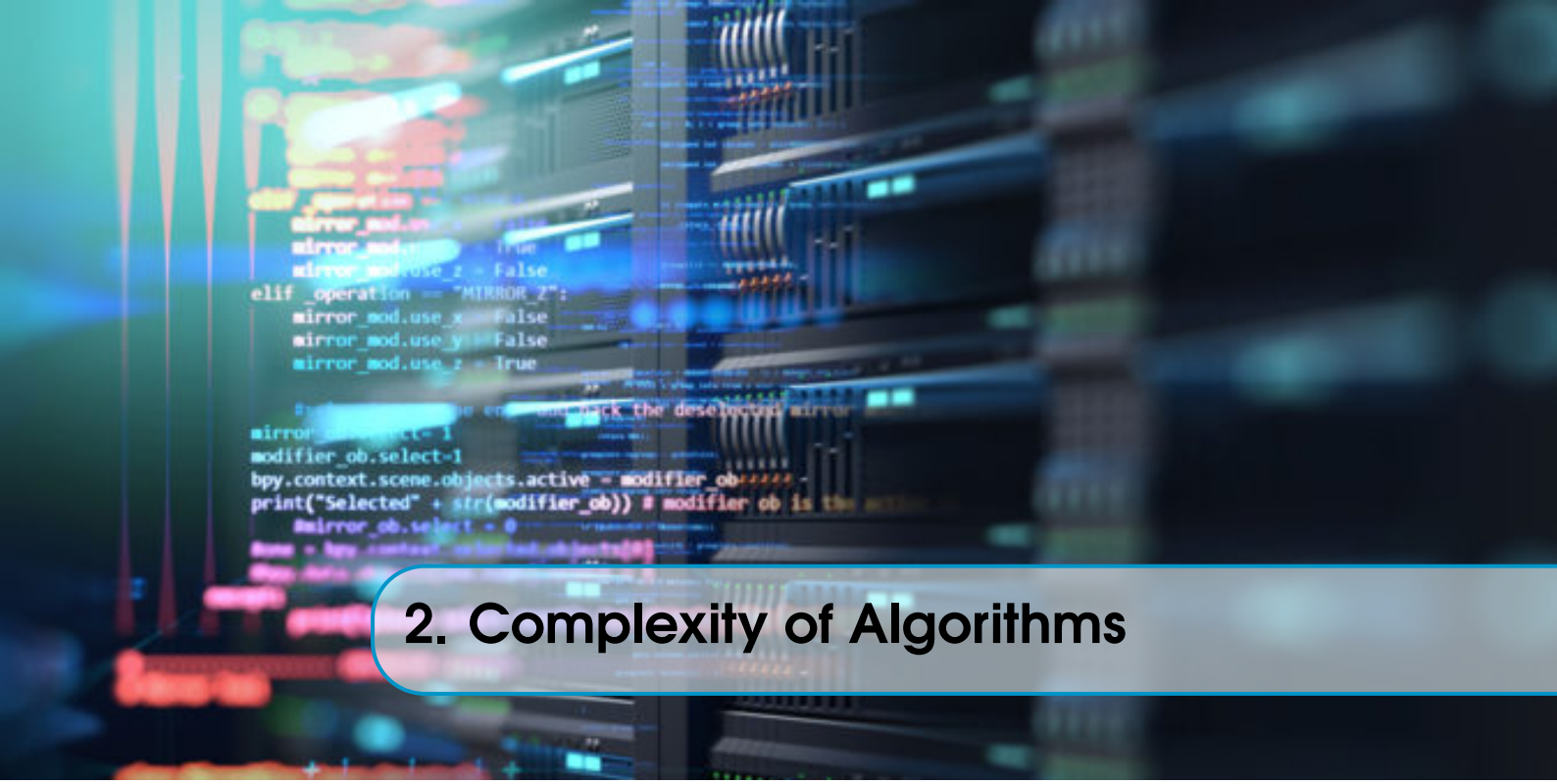
One might say that 'computing' became a scientific field in its own right, when the mathematics of real-world phenomena was asked to be constructive, that is, to go from proving the existence of solutions to actually obtaining them. At this point, algorithms become an object of study themselves, rather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

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

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

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

PART I
THEORY OF COMPUTATION



2. Complexity of Algorithms

Before we delve into the practicalities of optimising computer code (see chapters 4 and beyond), we must first tackle a number of conceptual difficulties that play a role in the determination whether the optimisation of computer code is even possible. In previous course we have already encountered the notion of Turing computable and the Church-Turing thesis of computability.

In this chapter, we will first focus briefly on functions that are not Turing computable (i.e., functions whose computation lies beyond the capabilities of computers), and follow in section 2.2 with the differentiations that one can make between all functions that are computable (that is to say, not all functions are equally computable). There, we also introduce the notion of algorithm and problem complexity as tools to indicate the 'difficulty' of computing the answer to a particular problem (or, likewise, the difficulty that a particular algorithm has in computing the answer to a particular problem).

2.1 A Noncomputable Function

As you might already know, a Turing machine, as introduced in a previous course, can be used to compute a number of function like, for instance, a successor function, which assigns each nonnegative integer input value n to the output value $n + 1$. We need merely place the input value in its binary form on the machine's tape, run the machine until it halts, and then read the output value from the tape. A function that can be computed in this manner by a Turing machine is said to be **Turing computable**.

Turing's conjecture was that the Turing-computable functions were the same as the computable functions. In other words, he conjectured that the

This section is loosely inspired on chapter 12.4 of *Computer Science: an Overview* by Brookshear [7].

computational power of Turing machines encompasses that of any algorithmic system or, equivalently, that (in contrast to such approaches as tables and algebraic formulas) the Turing machine concept provides a context in which solutions to all the computable functions can be expressed. Today, this conjecture is often referred to as the Church–Turing thesis, in reference to the contributions made by both Alan Turing and Alonzo Church. Since Turing's initial work, much evidence has been collected to support this thesis, and today the **Church–Turing thesis** is widely accepted. That is, the computable functions and the Turing-computable functions are considered the same.

The significance of this conjecture is that it gives insight to the capabilities and limitations of computing machinery. More precisely, it establishes the capabilities of Turing machines as a standard to which the powers of other computational systems can be compared. If a computational system is capable of computing all the Turing-computable functions, it is considered to be as powerful as any computational system can be.

We now identify an example of a function that is **not** Turing computable and so, by the Church-Turing thesis, is widely believed to be noncomputable in the general sense. Thus it is a function whose computation lies beyond the capabilities of computers. There are other kinds of non-computable functions, like Kolmogorov Complexity and the Busy Beaver problem.

2.1.1 The Halting Problem

The noncomputable function we are about to reveal is associated with a problem known as the **halting problem**, which (in an informal sense) is the problem of trying to predict in advance whether a program will terminate (or halt) if started under certain conditions. For example, consider the simple Bare Bones program

```
while X not 0 do;
   incr X
end;
```

If we execute this program with the initial value of `X` being zero, the loop will not be executed and the program's execution will quickly terminate. However, if we execute the program with any other initial value of `X`, the loop will be executed forever, leading to a nonterminating process

In this case, then, it is easy to conclude that the program's execution will halt only when it is started with `X` assigned the value zero. However, as we move to more complex examples, the task of predicting a program's behavior becomes more complicated. In fact, in some cases the task is impossible, as we shall see. But first we need to formalize our terminology and focus our thoughts more precisely.

Our example has shown that whether a program ultimately halts can depend on the initial values of its variables. Thus if we hope to predict whether a program's execution will halt, we must be precise in regard to these initial values. The choice we are about to make for these values might seem strange to you at first, but do not despair. Our goal is to take advantage of a technique

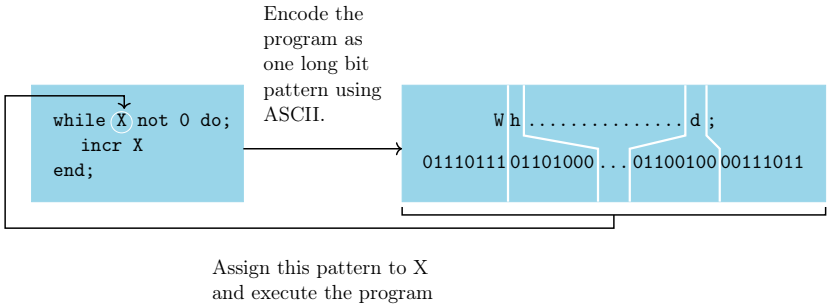


Figure 2.1 Testing a program for self-termination.

called **self-reference** – the idea of an object referring to itself. This ploy has repeatedly led to amazing results in mathematics from such informal curiosities as the sentence "This statement is false" to the more serious paradox represented by the question "Does the set of all sets contain itself?" What we are about to do, then, is set the stage for a line of reasoning similar to "If it does, then it doesn't; but, if it doesn't, then it does."

In our case self-reference will be achieved by assigning the variables in a program an initial value that represents the program itself. To this end, observe that each Bare Bones program can be encoded as a single long bit pattern in a one-character-per-byte format using ASCII, which can then be interpreted as the binary representation for a (rather large) nonnegative integer. It is this integer value that we assign as the initial value for the variables in the program.

Let us consider what would happen if we did this in the case of the simple program

```
while X not 0 do;
   incr X
end;
```

We want to know what would happen if we started this program with X assigned the integer value representing the program itself (Figure 2.1). In this case the answer is readily apparent. Because `X` would have a nonzero value, the program would become caught in the loop and never terminate. On the other hand, if we performed a similar experiment with the

```
clear X
while X not 0 do;
   incr X
end;
```

the program would terminate because the variable `X` would have the value zero by the time the `while-end` structure is reached regardless of its initial value.

Let us, then, make the following definition: A Bare Bones program is **self-terminating** if executing the program with all its variables initialized to the program's own encoded representation leads to a terminating process. Informally, a program is self-terminating if its execution terminates when started with itself as its input. Here, then, is the self-reference that we promised.

Note that whether a program is self-terminating probably has nothing to do with the purpose for which the program was written. It is merely a property that each Bare Bones program either possesses or does not possess. That is, each Bare Bones program is either self-terminating or not.

We can now describe the halting problem in a precise manner. It is the problem of determining whether Bare Bones programs are or are not self-terminating. We are about to see that there is no algorithm for answering this question in general. That is, there is no single algorithm that, when given any Bare Bones program, is capable of determining whether that program is or is not selfterminating. Thus the solution to the halting problem lies beyond the capabilities of computers.

The fact that we have apparently solved the halting problem in our previous examples and now claim that the halting problem is unsolvable might sound contradictory, so let us pause for clarification. The observations we used in our examples were unique to those particular cases and would not be applicable in all situations. What the halting problem requests is a single, generic algorithm that can be applied to any Bare Bones program to determine whether it is self-terminating. Our ability to apply certain isolated insights to determine whether a particular program is self-terminating in no way implies the existence of a single, generic approach that can be applied in all cases. In short, we might be able to build a machine that can solve a particular halting problem, but we cannot build a single machine that we could use to solve any halting problem that arises.

2.1.2 The Unsolvability of the Halting Problem

The following subsection presents a mathematical proof of why the halting problem is unsolvable. This is beyond the requirements of the course, and is only included for those interested.

We now want to show that solving the halting problem lies beyond the capabilities of machines. Our approach is to show that to solve the problem would require an algorithm for computing a noncomputable function. The inputs of the function in question are encoded versions of Bare Bones programs; its outputs are limited to the values 0 and 1. More precisely, we define the function so that an input representing a self-terminating program produces the output value 1 while an input representing a program that is not self-terminating produces the output value 0. For the sake of conciseness, we will refer to this function as the halting function.

Our task is to show that the halting function is not computable. Our approach is the technique known as "proof by contradiction." In short, we prove that a statement is false by showing that it cannot be true. Let us, then, show that the statement "the halting function is computable" cannot be true. Our entire argument is summarized in Figure 2.2.

If the halting function is computable, then (because Bare Bones is a universal programming language) there must be a Bare Bones program that computes it.

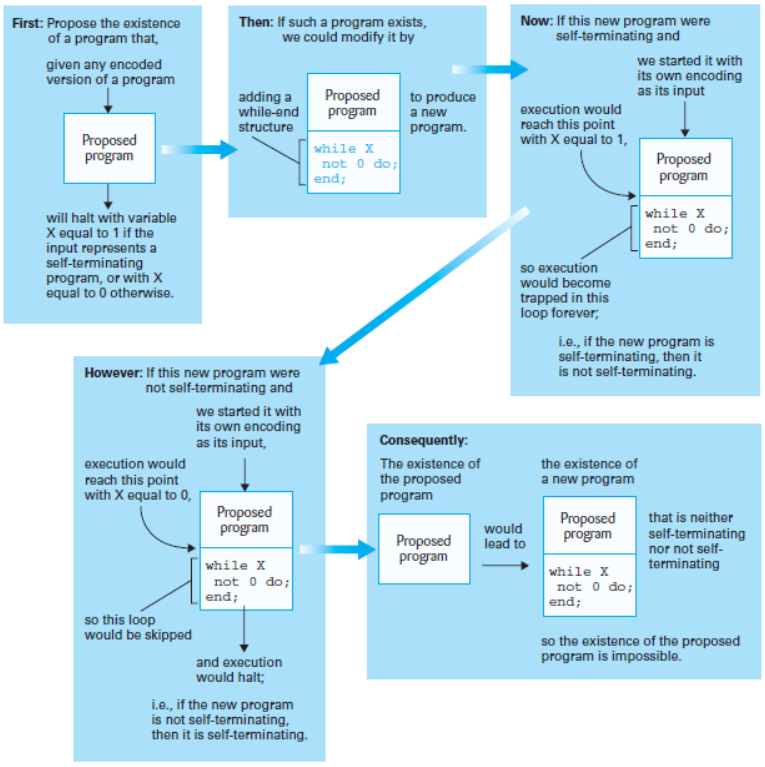


Figure 2.2 Proving the unsolvability of the halting problem.

In other words, there is a Bare Bones program that terminates with its output equal to 1 if its input is the encoded version of a self-terminating program and terminates with its output equal to 0 otherwise.

To apply this program we do not need to identify which variable is the input variable but instead merely to initialize all the program's variables to the encoded representation of the program to be tested. This is because a variable that is not an input variable is inherently a variable whose initial value does not affect the ultimate output value. We conclude that if the halting function is computable, then there is a Bare Bones program that terminates with its output equal to 1 if all its variables are initialized to the encoded version of a self-terminating program and terminates with its output equal to 0 otherwise.

Assuming that the program's output variable is named `X` (if it is not we could simply rename the variables), we could modify the program by attaching the statements

```
while X not 0 do;
end;
```

at its end, producing a new program. This new program must be either selfterminating or not. However, we are about to see that it can be neither.

In particular, if this new program were self-terminating and we ran it with its variables initialized to the program's own encoded representation, then when its execution reached the `while` statement that we added, the variable `X` would contain a 1. (To this point the new program is identical to the original program

that produced a 1 if its input was the representation of a selfterminating program.) At this point, the program's execution would be caught forever in the `while-end` structure because we made no provisions for `X` to be decremented within the loop. But this contradicts our assumption that the new program is self-terminating. Therefore we must conclude that the new program is not self-terminating.

If, however, the new program were not self-terminating and we executed it with its variables initialized to the program's own encoded representation, it would reach the added `while` statement with `X` being assigned the value 0. (This occurs because the statements preceding the `while` statement constitute the original program that produces an output of 0 when its input represents a program that is not self-terminating.) In this case, the loop in the `while-end` structure would be avoided and the program would halt. But this is the property of a self-terminating program, so we are forced to conclude that the new program is self-terminating, just as we were forced to conclude earlier that it is not self-terminating.

In summary, we see that we have the impossible situation of a program that on the one hand must be either self-terminating or not and on the other hand can be neither. Consequently, the assumption that led to this dilemma must be false.

We conclude that the halting function is not computable, and because the solution to the halting problem relies on the computation of that function we must conclude that solving the halting problem lies beyond the capabilities of any algorithmic system. Such problems are called **unsolvable problems**.

In closing, a major underlying questing is whether the powers of computing machines include those required for intelligence itself. While, machines can solve only problems with algoritmic solutions, we have now found that there are problems without algoritmic solutions. The question, then, is whether the human mind embodies more than the execution of algoritmic processes. If it does not, then the limits we have identified here are also limits of human thought. Needless to say, this is a highly debatable and sometimes emotional issue. If, for example, human minds are no more than programmed machines, then one would conclude that humans do not possess free will.

2.1.3 Exercises

Exercise 2.1

Is the following Bare Bones program self-terminating? Explain your answer.

```
incr X;
decr X;
```

Exercise 2.2

Is the following Bare Bones program self-terminating? Explain your answer.

```
copy X to Y;
incr Y;
incr Y;
while X not 0 do;
  decr X;
  decr X;
  decr Y;
  decr Y;
end;
decr Y;
while Y not 0 do;
end;
```

Exercise 2.3

What is wrong with the following scenario?

> In a certain community, everyone owns his or her own house. The house painter of the community claims to paint all those and only those houses that are not painted by their owners.

(*Hint*: Who paints the house painter's house?)

2.2 Complexity of Problems

In Section 2.1 we investigated the solvability of problems. In this section we are interested in the question of whether a solvable problem has a practical solution. We will find that some problems that are theoretically solvable are so complex that they are unsolvable from a practical point of view.

2.2.1 Efficiency of Algorithms

Search algorithms (like linear search or binary search) all accomplish the same goal – finding an element (or elements) matching a given search key if such an element exists. There are, however, several things that differentiate search algorithms from one another. The major difference is the amount of effort they require to complete the search. One way to describe this effort is with Θ notation ('big-theta'), which indicates how hard an algorithm may have to work to solve a problem. For search and sorting algorithms, this depends mainly on how many data elements there are.

$\Theta(1)$ Algorithms

Suppose an algorithm is designed to test whether the first element of an array is equal to the second. If the array has 10 elements, this algorithm requires one comparison. If the array has 1000 elements, it still requires one comparison.

This section is loosely inspired on chapters 5.6 and 12.4 of *Computer Science: an Overview* by Brookshear [7].

The algorithm is completely independent of the number of elements in the array. This algorithm is said to a constant run time, which is represented as $\Theta(1)$ and pronounced as "constant complexity". An algorithm that's $\Theta(1)$ does not necessarily require only one comparison. $\Theta(1)$ means that the number of comparisons is constant – it does not grow as the size of the array increases. An algorithm that tests whether the first element of an array is equal to any of the next three elements is still $\Theta(1)$ even though it requires three comparisons.

$\Theta(n)$ Algorithms

An algorithm that tests whether the first array element is equal to any of the other array elements requires at most $n-1$ comparisons, where n is the number of array elements. If the array has 10 elements, this algorithm requires up to nine comparisons. If the array has 1000 elements, it requires up to 999 comparisons. As n grows larger, the n part of the expression $n-1$ "dominates", and subtracting 1 becomes inconsequential. Big theta is designed to highlight these dominant terms and ignore terms that become unimportant as n grows. So, an algorithm that requires a total of $n-1$ comparisons (such as the one we described earlier) is said to be $\Theta(n)$. An $\Theta(n)$ algorithm is said to have a linear run time. $\Theta(n)$ is often pronounced "complexity on the order of n" or simply "linear complexity".

$\Theta(n2)$ Algorithms

Now suppose you have an algorithm that tests whether any element of an array is duplicated elsewhere in the array. The first element must be compared with every other element in the array. The second element must be compared with every other element except the first (it was already compared to the first). The third element must be compared with every other element except the first two. In the end, this algorithm makes $(n-1)+(n-2)+...+2+1$ or $\frac{n^2}{2}-\frac{n}{2}$ comparisons. As n increases, the n^2 term dominates, and the n term becomes inconsequential. Again, Big theta notation highlights the n^2 term, ignoring $\frac{n}{2}$.

Big theta is concerned with how an algorithm's run time grows in relation to the number of items processed. Suppose an algorithm requires n^2 comparisons. With four elements, the algorithm requires 16 comparisons; with eight elements, 64 comparisons. With this algorithm, doubling the number of elements quadruples the number of comparisons. Consider a similar algorithm requiring $\frac{n^2}{2}$ comparisons. With four elements, the algorithm requires eight comparisons; with eight elements, 32 comparisons. Again, doubling the number of elements quadruples the number of comparisons. Both of these algorithms grow as the square of n, so Θ ignores the constant, and both algorithms are considered to be $\Theta(n^2)$, which is referred to as quadratic run time and pronounced "complexity of n-squared" or more simply "quadratic complexity".

When n is small, $\Theta(n^2)$ algorithms (on today's computers) will not noticeably affect performance, but as n grows, you'll start to notice performance degradation. An $\Theta(n^2)$ algorithm running on a million-element array would require a trillion "operations" (where each could execute several machine instructions). We tested one such $\Theta(n^2)$ algorithms on a 100,000-element array using

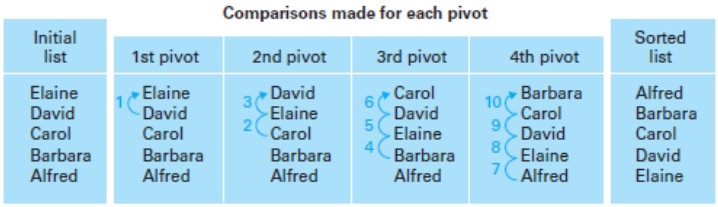


Figure 2.3 Applying the insertion sort in a worst-case situation

a current desktop computer, and it ran for many minutes. A billion-element array (not unusual in today's big data applications) could require a quintillion operations, which on that same desktop computer would take approximately 13.3 years to complete! As you'll see, $\Theta(n^2)$ algorithms, unfortunately, are easy to write. You'll also see algorithms with more favorable Θ measures. These more efficient algorithms often take a bit more cleverness and work to create, but their superior performance can be well worth the extra effort, especially as n gets large and as algorithms are integrated into larger programs.

In order to determine the complexity of a particular algorithm, one needs to perform a **complexity analysis**.

2.2.2 Complexity Analysis

Algorithm analysis often involves best-case, worst-case, and average-case scenarios. In general such analysis is performed in a generic context. That is, when considering algorithms for searching lists, we do not focus on a list of a particular length, but instead try to identify a formula that would indicate the algorithm's performance for lists of arbitrary lengths. For instance, when applied to a list with n entries, a sequential search algorithm will interrogate an average of $\frac{n}{2}$ entries, whereas a binary search algorithm will interrogate at most $lg\ n$ entries in its worst-case scenario. ($lg\ n$ represents the base two logarithm of n.)

Let us, as an example, analyse the insertion sort algorithm. Recall that this algorithm involves selecting a list entry, called the pivot entry, comparing this entry to those preceding it until the proper place for the pivot is found, and then inserting the pivot entry in this place. Since the activity of comparing two entries dominates the algorithm, our approach will be to count the number of such comparisons that are performed when sorting a list whose length is n.

The algorithm begins by selecting the second list entry to be the pivot. It then progresses by picking successive entries as the pivot until it has reached the end of the list. In the best possible case, each pivot is already in its proper place, and thus it needs to be compared to only a single entry before this is discovered. Thus, in the best case, applying the insertion sort to a list with n entries requires $n - 1$ comparisons. (The second entry is compared to one entry, the third entry to one entry, and so on.)

In contrast, the worst-case scenario is that each pivot must be compared to all the preceding entries before its proper location can be found. This occurs if the original list is in reverse order. In this case the first pivot (the second

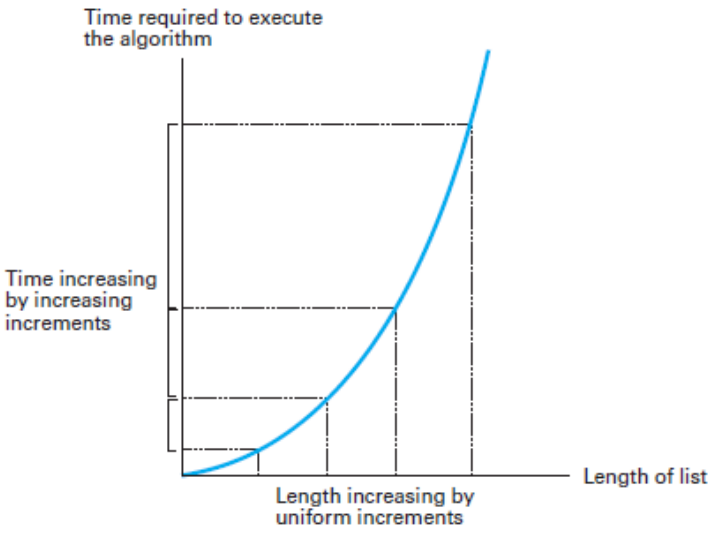


Figure 2.4 Graph of worst-case analysis of the insertion sort algorithm.

list entry) is compared to one entry, the second pivot (the third list entry) is compared to two entries, and so on (Figure 2.3). Thus the total number of comparisons when sorting a list of n entries is $1 + 2 + 3 + ... + (n + 1)$, which is equivalent to $\frac{1}{2}(n^2 - n)$. In particular, if the list contained 10 entries, the worst-case scenario of the insertion sort algorithm would require 45 comparisons.

In the average case of the insertion sort, we would expect each pivot to be compared to half of the entries preceding it. This results in half as many comparisons as were performed in the worst case, or a total of $\frac{1}{4}(n^2 - n)$ comparisons to sort a list of n entries. If, for example, we use the insertion sort to sort a variety of lists of length 10, we expect the average number of comparisons per sort to be 22.5.

The significance of these results is that the number of comparisons made during the execution of the insertion sort algorithm gives an approximation of the amount of time required to execute the algorithm. Using this approximation, Figure 2.4 shows a graph indicating how the time required to execute the insertion sort algorithm increases as the length of the list increases. This graph is based on our worst-case analysis of the algorithm, where we concluded that sorting a list of length n would require at most $\frac{1}{2}(n^2 - n)$ comparisons between list entries. On the graph, we have marked several list lengths and indicated the time required in each case. Notice that as the list lengths increase by uniform increments, the time required to sort the list increases by increasingly greater amounts. Thus the algorithm becomes less efficient as the size of the list increases.

The distinguishing factor between Figures 2.4 and, for instance, the graph of the run time requirements of binary search (which is logarithmic) is the general shape of the graphs involved. This general shape reveals how well an algorithm should be expected to perform for larger and larger inputs. Moreover, the general shape of a graph is determined by the type of the expression being

represented rather than the specifics of the expression—all linear expressions produce a straight line; all quadratic expressions produce a parabolic curve; all logarithmic expressions produce the logarithmic shape. It is customary to identify a shape with the simplest expression that produces that shape. In particular, we identify the parabolic shape with the expression n^2 and the logarithmic shape with the expression lgn.

Since the shape of the graph obtained by comparing the time required for an algorithm to perform its task to the size of the input data reflects the efficiency characteristics of the algorithm, it is common to classify algorithms according to the shapes of these graphs—normally based on the algorithm's worst-case analysis. The notation used to identify these classes is sometimes called big-theta notation. All algorithms whose graphs have the shape of a parabola, such as the insertion sort, are put in the class represented by $\Theta(n^2)$ (read "big theta of n squared"); all algorithms whose graphs have the shape of a logarithmic expression, such as the binary search, fall in the class represented by $\Theta(lg\ n)$ (read "big theta of log n"). Knowing the class in which a particular algorithm falls allows us to predict its performance and to compare it against other algorithms that solve the same problem. Two algorithms in $\Theta(n^2)$ will exhibit similar changes in time requirements as the size of the inputs increases. Moreover, the time requirements of an algorithm in $\Theta(lg\ n)$ will not expand as rapidly as that of an algorithm in $\Theta(n^2)$.

2.2.3 Measuring a Problem's Complexity

In the determination of the efficiency of algorithms before, we used big-theta notation to classify algorithms according to the time required to execute them. We found that the insertion sort algorithm is in the class $\Theta(n^2)$. The complexity of sequential search algorithm is in $\Theta(n)$, and that of binary search algorithm is in $\Theta(lg\ n)$. We now use this classification system to help us identify the **complexity of problems**. Our goal is to develop a classification system that tells us which problems are more complex than others and ultimately which problems are so complex that their solutions lie beyond practicality.

The reason that our present study is based on our knowledge of algorithm efficiency is that we wish to measure the complexity of a problem in terms of the complexity of its solutions. We consider a simple problem to be one that has a simple solution; a complex problem is one that does not have a simple solution. Note that the fact that a problem has a difficult solution does not necessarily mean that the problem itself is complex. After all, a problem has many solutions, one of which is bound to be complex. Thus to conclude that a problem itself is complex requires that we show that none of its solutions are simple.

In computer science, the problems of interest are those that are solvable by machines. The solutions to these problems are formulated as algorithms. Thus the complexity of a problem is determined by the properties of the algorithms that solve that problem. More precisely, the complexity of the simplest algorithm for solving a problem is considered to be the complexity of

the problem itself.

But how do we measure the *complexity* of an algorithm? Unfortunately, the term complexity has different interpretations. One deals with the amount of decision making and branching involved in the algorithm. In this light, a complex algorithm would be one that involves a twisted, entwined set of directions. This interpretation might be compatible with the point of view of a software engineer who is interested in issues relating to algorithm discovery and representation, but it does not capture the concept of complexity from a machine's point of view. A machine does not really make decisions when selecting the next instruction for execution but merely follows its machine cycle over and over, each time executing the instruction that is indicated by the program counter. Consequently, a machine can execute a set of tangled instructions as easily as it can execute a list of instructions in a simple sequential order. This interpretation of complexity, therefore, tends to measure the difficulty encountered in an algorithm's representation rather than the complexity of the algorithm itself.

An interpretation that more accurately reflects the complexity of an algorithm from a machine's point of view is to measure the number of steps that must be performed when executing the algorithm. Note that this is not the same as the number of instructions appearing in the written program. A loop whose body consists of a single statement but whose control requests the body's execution 100 times is equivalent to 100 statements when executed. Such a routine is therefore considered more complex than a list of 50 individually written statements, even though the latter appears longer in written form. The point is that this meaning of complexity is ultimately concerned with the time it takes a machine to execute a solution and not with the size of the program representing the solution.

We therefore consider a problem to be complex if all its solutions require a lot of time. This definition of complexity is referred to as time complexity. We have already met the concept of time complexity indirectly through our study of algorithm efficiency in Section 2.2.2. After all, the study of an algorithm's efficiency is the study of the algorithm's time complexity—the two are merely inverses of each other. That is, "more efficient" equals "less complex." Thus, in terms of time complexity, the sequential search algorithm (which we found to be in $\Theta(n)$) is a more complex solution to the problem of searching a list than is the binary search algorithm (which we found to be in $\Theta(lg\ n)$).

Let us now apply our knowledge of the complexity of algorithms to obtain a means of identifying the complexity of problems. We define the (time) complexity of a problem to be $\Theta(f(n))$, where f(n) is some mathematical expression in n, if there is an algorithm for solving the problem whose time complexity is in $\Theta(f(n))$ and no other algorithm for solving the problem has a lower time complexity. That is, the (time) complexity of a problem is defined to be the (time) complexity of its best solution. Unfortunately, finding the best solution to a problem and knowing that it is the best is often a difficult problem in itself. In such situations, a variation of big-theta notation called **big O notation** (pronounced "big oh notation") is used to represent what is

Figure 2.5 A procedure MergeLists for merging two lists.

```
procedure MergeLists (InputListA, InputListB, OutputList)

if (both input lists are empty) then (Stop, with OutputList empty)
if (InputListA is empty)
  then (Declare it to be exhausted)
  else (Declare its first entry to be its current entry)
if (InputListB is empty)
  then (Declare it to be exhausted)
  else (Declare its first entry to be its current entry)
while (neither input list is exhausted) do
    (Put the "smaller" current entry in OutputList;
     if (that current entry is the last entry in its corresponding input list)
        then (Declare that input list to be exhausted)
        else (Declare the next entry in that input list to be the list's current entry )
    )
Starting with the current entry in the input list that is not exhausted,
    copy the remaining entries to OutputList.
```

known about a problem's complexity. More precisely, if $f(n)$ is a mathematical expression in n and if a problem can be solved by an algorithm in $\Theta(f(n))$ then we say that the problem is in $O(f(n))$ (pronounced "big oh of $f(n)$"). Thus, to say that a problem belongs to $O(f(n))$ means that it has a solution whose complexity is in $\Theta(f(n))$ but it could possibly have a better solution.

Our investigation of searching and sorting algorithms tells us that the problem of searching within a list of length n (when all we know is that the list has previously been sorted) is in $O(lg\ n)$ because the binary search algorithm solves the problem. Moreover, researchers have shown that the searching problem is actually in $\Theta(lg\ n)$ so the binary search represents an optimal solution for that problem. In contrast, we know that the problem of sorting a list of length n (when we know nothing about the original distribution of the values in it) is in $O(n^2)$ because the insertion sort algorithm solves the problem. The problem of sorting, however, is known to be in $\Theta(n\ lg\ n)$, which tells us that the insertion sort algorithm is not an optimal solution (in the context of time complexity).

An example of a better solution to the sorting problem is the merge sort algorithm. Its approach is to merge small, sorted portions of the list to obtain larger sorted portions that can then be merged to obtain still larger sorted portions. Each merging process applies the merge algorithm that we encountered when discussing sequential files (Figure 2.5). The complete (recursive) merge sort algorithm is presented as the procedure called MergeSort in Figure 2.6. When asked to sort a list, this procedure first checks to see if the list is shorter than two entries. If so, the procedure's task is complete. If not, the procedure divides the list into two pieces, asks other copies of the procedure MergeSort to sort these pieces, and then merges these sorted pieces together to obtain the final sorted version of the list.

To analyze the complexity of this algorithm, we first consider the number of comparisons between list entries that must be made in merging a list of length r with a list of length s. The merge process proceeds by repeatedly comparing an entry from one list with an entry from the other and placing the "smaller" of the two entries in the output list. Thus each time a comparison is made, the

```
procedure MergeSort (List)

if (List has more than one entry)
 then (Apply the procedure MergeSort to sort the first half of List;
       Apply the procedure MergeSort to sort the second half of List;
       Apply the procedure MergeLists to merge the first and second
          halves of List to produce a sorted version of List
       )
```

Figure 2.6 The merge sort algorithm implemented as a procedure MergeSort.

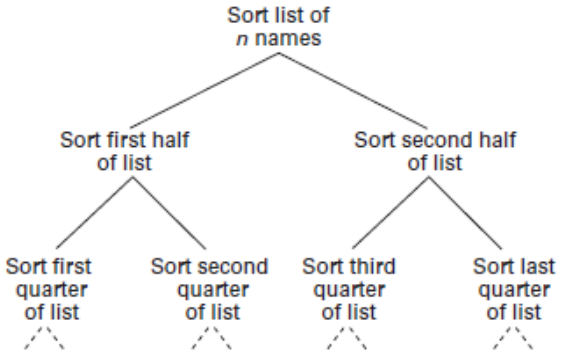


Figure 2.7 The hierarchy of problems generated by the merge sort algorithm.

number of entries still to be considered is reduced by one. Because there are only $r + s$ entries to begin with, we conclude that the process of merging the two lists will involve no more than $r + s$ comparisons.

We now consider the entire merge sort algorithm. It attacks the task of sorting a list of length n in such a way that the initial sorting problem is reduced to two smaller problems, each of which is asked to sort a list of length approximately $n/2$. These two problems are in turn reduced to a total of four problems of sorting lists of length approximately $n/4$. This division process can be summarized by the tree structure in Figure 2.7, where each node of the tree represents a single problem in the recursive process and the branches below a node represent the smaller problems derived from the parent. Hence, we can find the total number of comparisons that occur in the entire sorting process by adding together the number of comparisons that occur at the nodes in the tree.

Let us first determine the number of comparisons made across each level of the tree. Observe that each node appearing across any level of the tree has the task of sorting a unique segment of the original list. This is accomplished by the merge process and therefore requires no more comparisons than there are entries in the list segment, as we have already argued. Hence, each level of the tree requires no more comparisons than the total number of entries in the list segments, and because the segments across a given level of the tree represent disjoint portions of the original list, this total is no greater than the length of the original list. Consequently, each level of the tree involves no more than n comparisons. (Of course the lowest level involves sorting lists of length less than two, which involves no comparisons at all.)

Now we determine the number of levels in the tree. For this, observe that the process of dividing problems into smaller problems continues until lists of length less than two are obtained. Thus the number of levels in the tree is

determined by the number of times that, starting with the value n, we can repeatedly divide by two until the result is no larger than one, which is $lg\ n$. More precisely, there are no more than $\lceil lg\ n \rceil$ levels of the tree that involve comparisons, where the notation $\lceil lg\ n \rceil$ represents the value of $lg\ n$ rounded up to the next integer.

Finally, the total number of comparisons made by the merge sort algorithm when sorting a list of length n is obtained by multiplying the number of comparisons made at each level of the tree by the number of levels in which comparisons are made. We conclude that this is no larger than $n\ \lceil lg\ n \rceil$ Because the graph of $n\ \lceil lg\ n \rceil$ has the same general shape as the graph of $n\ lg\ n$, we conclude that the merge sort algorithm belongs to $O(n\ lg\ n)$. Combining this with the fact that researchers tell us that the sorting problem has complexity $\Theta(n\ lg\ n)$ implies that the merge sort algorithm represents an optimal solution to the sorting problem.

Aside 2.1 — Space complexity. An alternative to measuring complexity in terms of time is to measure storage space requirements instead—resulting in a measure known as **space complexity**. That is, the space complexity of a problem is determined by the amount of storage space required to solve the problem. In the text we have seen that the time complexity of sorting a list with n entries is $O(n\ lg\ n)$. The space complexity of the same problem is no more than $O(n + 1) = O(n)$. After all, sorting a list with n entries using the insertion sort requires space for the list itself plus space to store a single entry on a temporary basis. Thus, if we were asked to sort longer and longer lists, we would find that the time required for each task would increase more rapidly than the space required. This is in fact a common phenomenon. Because it takes time to use space, a problem's space complexity never grows more rapidly than its time complexity.

There are often tradeoffs made between time and space complexity. In some applications it might be advantageous to perform certain computations in advance and store the results in a table from which they can be retrieved quickly when needed. Such a "table lookup" technique decreases the time required to obtain a result once it is actually needed, at the expense of the additional space required by the table. On the other hand, data compression is often used to reduce storage requirements at the expense of the additional time required to compress and decompress the data.

2.2.4 Polynomial Versus Nonpolynomial Problems

Suppose $f(n)$ and $g(n)$ are mathematical expressions. To say that $g(n)$ is bounded by $f(n)$ means that as we apply these expressions to larger and larger values of n, the value of $f(n)$ will ultimately become greater than that of $g(n)$ and remain greater than $g(n)$ for all larger values of n. In other words, that $g(n)$ is bounded by $f(n)$ means that the graph of $f(n)$ will be above the graph of $g(n)$ for "large" values of n. For instance, the expression $lg\ n$ is bounded by the expression n (Figure 2.8a), and $n\ lg\ n$ is bounded by n^2 (Figure 2.8b).

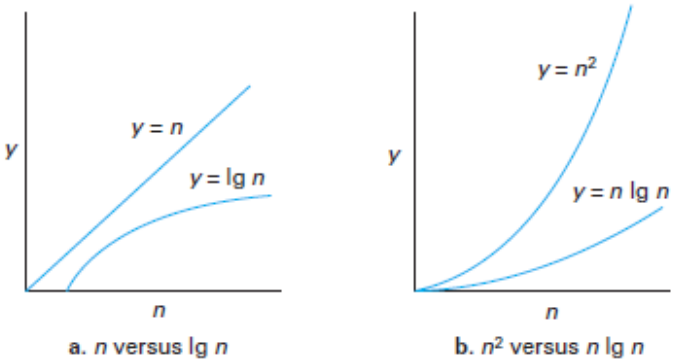


Figure 2.8 Graphs of the mathematical expressions n, $lg\ n$, $n\ lg\ n$, and n^2.

We say that a problem is a **polynomial problem** if the problem is in $O(f(n))$, where the expression $f(n)$ is either a polynomial itself or bounded by a polynomial. The collection of all polynomial problems is represented by **P**. Note that our previous investigations tell us that the problems of searching a list and of sorting a list belong to P.

To say that a problem is a polynomial problem is a statement about the time required to solve the problem. We often say that a problem in P can be solved in polynomial time or that the problem has a polynomial time solution.

Identifying the problems that belong to P is of major importance in computer science because it is closely related to questions regarding whether problems have practical solutions. Indeed, problems that are outside the class P are characterized as having extremely long execution times, even for inputs of moderate size. Consider, for example, a problem whose solution requires 2^n steps. The exponential expression 2^n is not bounded by any polynomial—if $f(n)$ is a polynomial, then as we increase the value of n, we will find that the values of 2n will ultimately be larger than those of $f(n)$. This means that an algorithm with complexity $\Theta(2^n)$ will generally be less efficient, and thus require more time, than an algorithm with complexity $\Theta(f(n))$. An algorithm whose complexity is identified by an exponential expression is said to require exponential time.

As a particular example, consider the problem of listing all possible subcommittees that can be formed from a group of n people. Because there are $2^n - 1$ such subcommittees (we allow a subcommittee to consist of the entire group but do not consider the empty set to be a subcommittee), any algorithm that solves this problem must have at least $2^n - 1$ steps and thus a complexity at least that large. But, the expression $2^n - 1$, being an exponential expression, is not bounded by any polynomial. Hence any solution to this problem becomes enormously time-consuming as the size of the group from which the committees are selected increases.

In contrast to our subcommittee problem, whose complexity is large merely because of the size of its output, problems exist whose complexities are large even though their ultimate output is merely a simple yes or no answer. An

example involves the ability to answer questions about the truth of statements involving the addition of real numbers. For instance, we can easily recognize that the answer to the question "Is it true that there is a real number that when added to itself produces the value 6?" is yes, whereas the answer to "Is it true that there is a nonzero real number which when added to itself is 0?" is no. However, as such questions become more involved, our ability to answer them begins to fade. If we found ourselves faced with many such questions, we might be tempted to turn to a computer program for assistance. Unfortunately, the ability to answer these questions has been shown to require exponential time, so even a computer ultimately fails to produce answers in a timely manner as the questions become more involved.

The fact that the problems that are theoretically solvable but are not in P have such enormous time complexities leads us to conclude that these problems are essentially unsolvable from a practical point of view. Computer scientists call these problems **intractable**. In turn, the class P has come to represent an important boundary that distinguishes intractable problems from those that might have practical solutions. Thus an understanding of the class P has become an important pursuit within computer science.

2.2.5 NP Problems

Let us now consider the **traveling salesman problem**, which involves a traveling salesman who must visit each of his clients in different cities without exceeding his travel budget. His problem, then, is to find a path (starting from his home, connecting the cities involved, and returning to his home) whose total length does not exceed his allowed mileage.

The traditional solution to this problem is to consider the potential paths in a systematic manner, comparing the length of each path to the mileage limit until either an acceptable path is found or all possibilities have been considered. This approach, however, does not produce a polynomial time solution. As the number of cities increases, the number of paths to be tested grows more rapidly than any polynomial. Thus, solving the traveling salesman problem in this manner is impractical for cases involving large numbers of cities.

We conclude that to solve the traveling salesman problem in a reasonable amount of time, we must find a faster algorithm. Our appetite is whetted by the observation that if a satisfactory path exists and we happen to select it first, our present algorithm terminates quickly. In particular, the following list of instructions can be executed quickly and has the potential of solving the problem:

```
Pick one of the possible paths, and compute its total
 distance.
If (this distance is not greater than the allowable mileage)
     then (declare a success)
     else (declare nothing)
```

However, this set of instructions is not an algorithm in the technical sense.

Its first instruction is ambiguous in that it does not specify which path is to be selected nor does it specify how the decision is to be made. Instead it relies on the creativity of the mechanism executing the program to make the decision on its own. We say that such instructions are nondeterministic, and we call an "algorithm" containing such statements a **nondeterministic algorithm**.

Note that as the number of cities increases, the time required to execute the preceding nondeterministic algorithm grows relatively slowly. The process of selecting a path is merely that of producing a list of the cities, which can be done in a time proportional to the number of cities involved. Moreover, the time required to compute the total distance along the chosen path is also proportional to the number of cities to be visited, and the time required to compare this total to the mileage limit is independent of the number of cities. In turn, the time required to execute the nondeterministic algorithm is bounded by a polynomial. Thus it is possible to solve the traveling salesman problem by a nondeterministic algorithm in polynomial time.

Of course, our nondeterministic solution is not totally satisfactory. It relies on a lucky guess. But its existence is enough to suggest that perhaps there is a deterministic solution to the traveling salesman problem that runs in polynomial time. Whether or not this is true remains an open question. In fact, the traveling salesman problem is one of many problems that are known to have nondeterministic solutions that execute in polynomial time but for which no deterministic polynomial time solution has yet been found. The tantalizing efficiency of the nondeterministic solutions to these problems causes some to hope that efficient deterministic solutions will be found someday, yet most believe that these problems are just complex enough to escape the capabilities of efficient deterministic algorithms.

A problem that can be solved in polynomial time by a nondeterministic algorithm is called a **nondeterministic polynomial problem**, or an **NP problem** for short. It is customary to denote the class of NP problems by **NP**. Note that all the problems in P are also in NP, because any (deterministic) algorithm can have a nondeterministic instruction added to it without affecting its performance.

Whether all of the NP problems are also in P, however, is an open question, as demonstrated by the traveling salesman problem. This is perhaps the most widely known unsolved problem in computer science today. Its solution could have significant consequences. For example, in the next section we will learn that encryption systems have been designed whose integrity relies on the enormous time required to solve problems similar to the traveling salesman problem. If it turns out that efficient solutions to such problems exist, these encryption systems will be compromised.

Efforts to resolve the question of whether the class NP is, in fact, the same as the class P have led to the discovery of a class of problems within the class NP known as the **NP-complete problems**. These problems have the property that a polynomial time solution for any one of them would provide a polynomial time solution for all the other problems in NP as well. That is, if

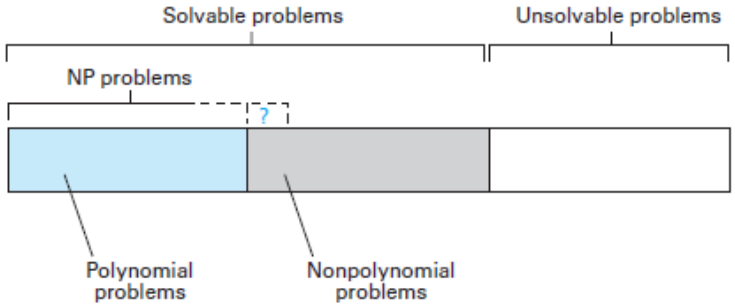


Figure 2.9 A graphic summation of the problem classification.

a (deterministic) algorithm can be found that solves one of the NP-complete problems in polynomial time, then that algorithm can be extended to solve any other problem in NP in polynomial time. In turn, the class NP would be the same as the class P. The traveling salesman problem is an example of an NP-complete problem.

In summary, we have found that problems can be classified as either solvable (having an algorithmic solution) or unsolvable (not having an algorithmic solution), as depicted in Figure 2.9. Moreover, within the class of solvable problems are two subclasses. One is the collection of polynomial problems that contains those problems with practical solutions. The second is the collection of nonpolynomial problems whose solutions are practical for only relatively small or carefully selected inputs. Finally, there are the mysterious NP problems that thus far have evaded precise classification.

Aside 2.2 — Deterministic versus Nondeterministic. In many cases, there is a fine line between a deterministic and a nondeterministic "algorithm." However, the distinction is quite clear and significant. A deterministic algorithm does not rely on the creative capabilities of the mechanism executing the algorithm, whereas a nondeterministic "algorithm" might. For instance, compare the instruction

> Go to the next intersection and turn either right or left.

and the instruction

> Go to the next intersection and turn right or left depending on what the person standing on the corner tells you to do.

In either case the action taken by the person following the directions is not determined prior to actually executing the instruction. However, the first instruction requires the person following the directions to make a decision based on his or her own judgment and is therefore nondeterministic. The second instruction makes no such requirements of the person following the directions—the person is told what to do at each stage. If several different people follow the first instruction, some might turn right, while others might

turn left. If several people follow the second instruction and receive the same information, they will all turn in the same direction. Herein lies an important distinction between deterministic and nondeterministic "algorithms." If a deterministic algorithm is executed repeatedly with the same input data, the same actions will be performed each time. However, a nondeterministic "algorithm" might produce different actions when repeated under identical conditions.

2.2.6 Exercises

Exercise 2.4
Suppose a problem can be solved by an algorithm in $\Theta(2^n)$. What can we conclude about the complexity of the problem?

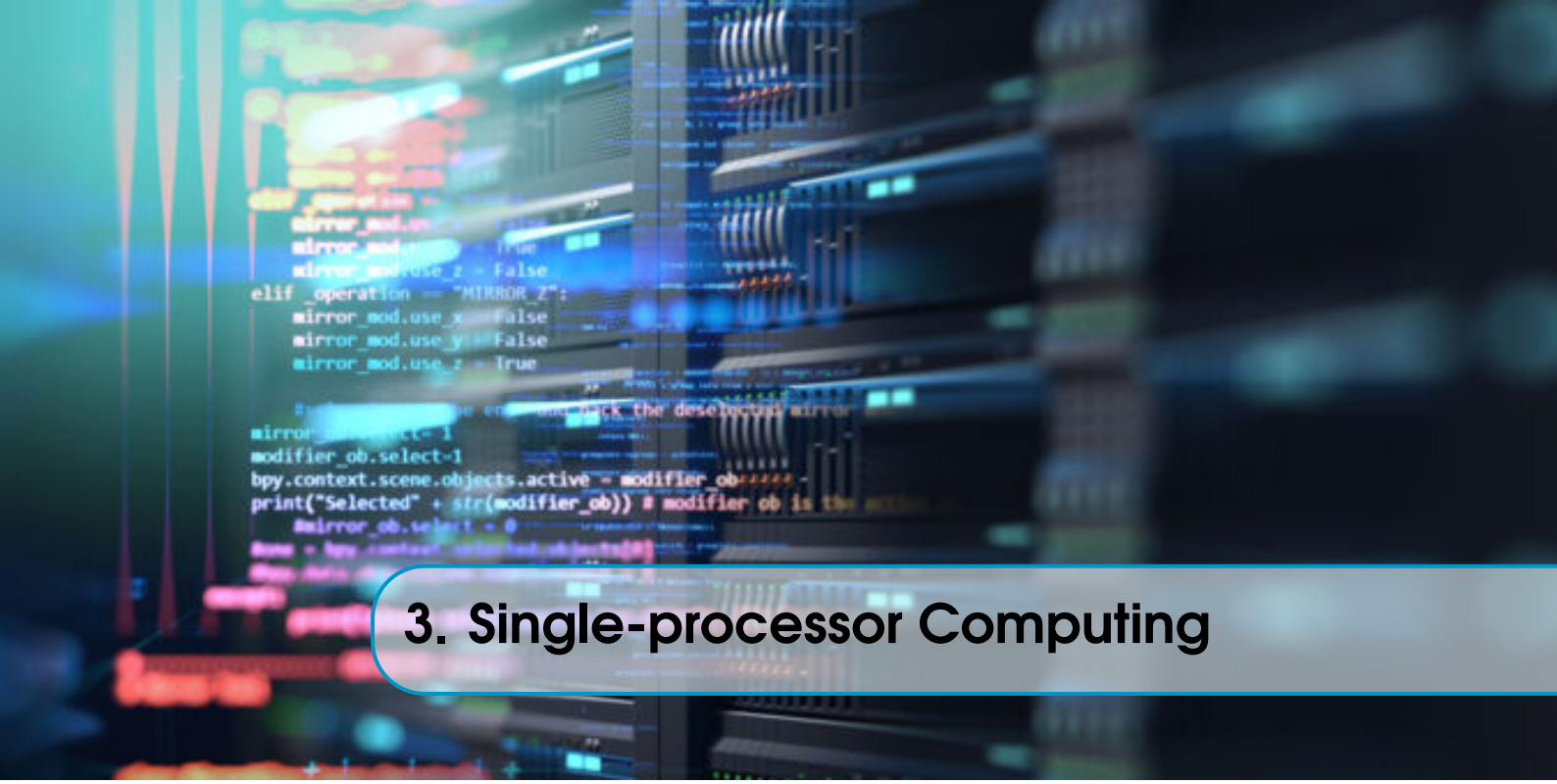
Exercise 2.5
Suppose a problem can be solved by an algorithm in $\Theta(n^2)$ as well as another algorithm in $\Theta(2^n)$. Will one algorithm always outperform the other?

Exercise 2.6
List all of the subcommittees that can be formed from a committee consisting of the two members Alice and Bill. List all the subcommittees that can be formed from the committee consisting of Alice, Bill, and Carol. What about the subcommittees from Alice, Bill, Carol, and David?

Exercise 2.7
Give an example of a polynomial problem. Give an example of a nonpolynomial problem. Give an example of an NP problem that as yet has not been shown to be a polynomial problem.

Exercise 2.8
If the complexity of algorithm X is greater than that of algorithm Y, is algorithm X necessarily harder to understand than algorithm Y? Explain your answer.

PART II
THEORY OF PARALLELISATION



3. Single-processor Computing

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

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

In this chapter, we will focus on what goes on inside a CPU and its memory system. We start with a brief general discussion of how instructions are handled, then we will look into the arithmetic processing in the processor core.

This chapter will give you a basic understanding of the issues involved in CPU design, how it affects performance.For much more detail, see an online book about PC architecture [31], and the standard work about computer architecture, Hennessy and Patterson [26].

3.1 The Von Neumann architecture

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

cycle'[1].

This setup distinguishes modern processors for the very earliest, and some special purpose contemporary, designs where the program was hard-wired. It also allows programs to modify themselves or generate other programs, since instructions and data are in the same storage. This allows us to have editors and compilers: the computer treats program code as data to operate on [2]. In this book we will not explicitly discuss compilers, the programs that translate high level languages to machine instructions. However, on occasion we will discuss how a program at high level can be written to ensure efficiency at the low level.

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

The machine instructions that a processor executes, as opposed to the higher level languages users write in, typically specify the name of an operation, as well as of the locations of the operands and the result. These locations are not expressed as memory locations, but as *registers*: a small number of named memory locations that are part of the CPU [3]. As an example, here is a simple C routine

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

and its X86 assembler output, obtained by[4] `gcc -O2 -S -o - store.c`:

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

[1] This model with a prescribed sequence of instructions is also referred to as *control flow*.

[2] At one time, the stored program concept was included as an essential component the ability for a running program to modify its own source. However, it was quickly recognized that this leads to unmaintainable code, and is rarely done in practice [15].

[3] Direct-to-memory architectures are rare, though they have existed. The Cyber 205 supercomputer in the 1980s could have three data streams, two from memory to the processor, and one back from the processor to memory, going on at the same time. Such an architecture is only feasible if memory can keep up with the processor speed, which is no longer the case these days.

[4] This is 64-bit output; add the option `-m64` on 32-bit systems.

The instructions here are:

- A load from memory to register (line 6);
- Another load, combined with an addition (line 7);
- Writing back the result to memory (line 8).

Each instruction is processed as follows:

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

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

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

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

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

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

3.2 Modern processors

Modern processors are quite complicated, and in this section we will give a short tour of what their constituent parts. Figure 3.1 is a picture of the *die* of an *Intel*Sandy Bridge processor. This chip is about an inch in size and contains close to a billion transistors.

3.2.1 The processing cores

In the Von Neuman model there is a single entity that executes instructions. This has not been the case in increasing measure since the early 2000s. The Sandy Bridge pictured in figure 3.1 has eight *cores*, each of which is an independent unit executing a stream of instructions. In this chapter we will mostly discuss aspects of a single *core*; section 3.4 will discuss the integration aspects of the multiple cores.

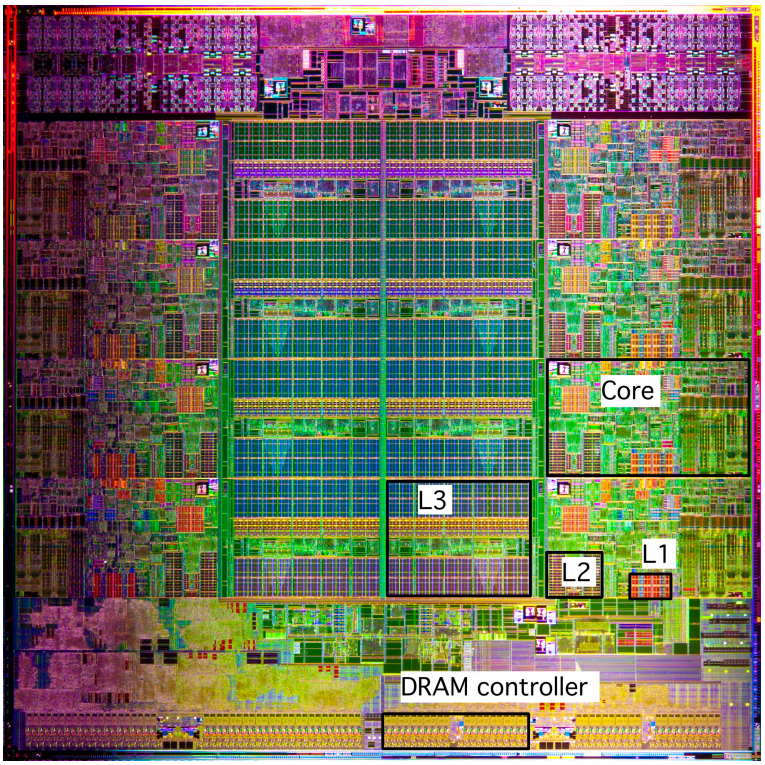


Figure 3.1 The Intel Sandy Bridge processor die

Instruction handling

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

In the block diagram (figure 3.2) you see various units that are concerned with instruction handling: This cleverness actually costs considerable energy,

as well as sheer amount of transistors. For this reason, processors such as the first generation Intel Xeon Phi, the *Knights Corner*, used *in-order* instruction handling. However, in the next generation, the *Knights Landing*, this decision was reversed for reasons of performance.

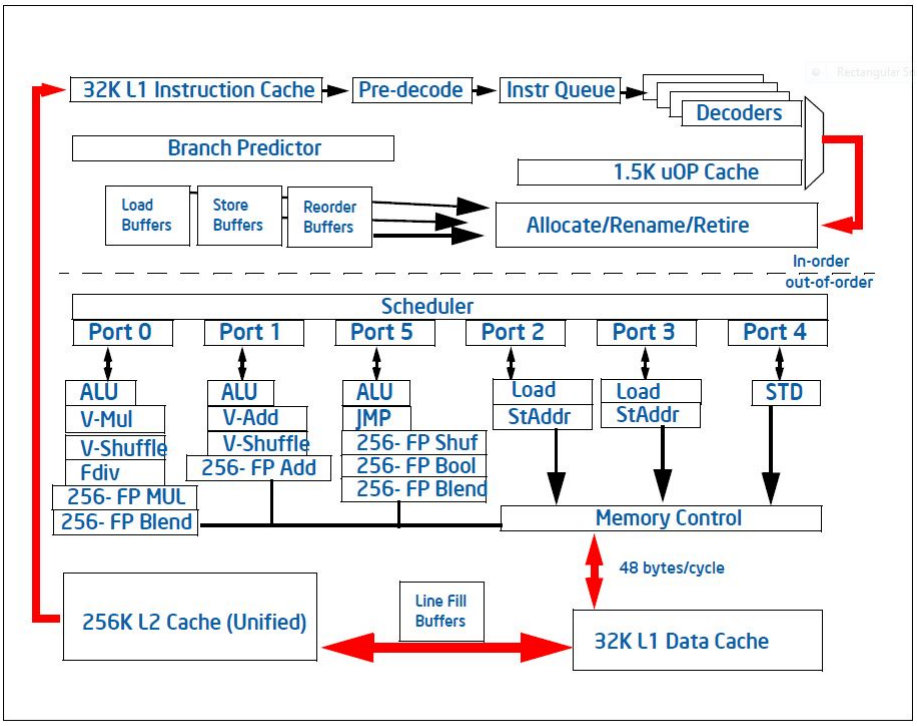


Figure 2-1. Intel microarchitecture code name Sandy Bridge Pipeline Functionality

Figure 3.2 Block diagram of the Intel Sandy Bridge core

Floating point units

R This section, and the remainder of this book assumes at least a basic understanding of the encoding of floating-point numbers. Appendix B contains a brief fresh-up, if needed.

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

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

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

Another way to increase performance is to have a *Fused Multiply-Add* (FMA) unit, which can execute the instruction $x \leftarrow ax + b$ in the same amount of time

as a separate addition or multiplication. Together with pipelining (see below), this means that a processor has an asymptotic speed of several floating point operations per clock cycle.

Table 3.1 Floating point capabilities (per core) of several processor architectures, and DAXPY cycle number for 8 operands

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

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

Pipelining

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

The idea behind a pipeline is as follows. Assume that an operation consists of multiple simpler operations, and that for each suboperation there is separate hardware in the processor. For instance, an addition instruction can have the following components:

- Decoding the instruction, including finding the locations of the operands.
- Copying the operands into registers ('data fetch').
- Aligning the exponents; the addition $.35 \times 10^{-1} + .6 \times 10^{-2}$ becomes $.35 \times 10^{-1} + .06 \times 10^{-1}$.
- Executing the addition of the mantissas, in this case giving .41.
- Normalizing the result, in this example to $.41 \times 10^{-1}$. (Normalization in this example does not do anything. Check for yourself that in $.3 \times 10^{0} + .8 \times 10^{0}$ and $.35 \times 10^{-3} + (-.34) \times 10^{-3}$ there is a non-trivial adjustment.)
- Storing the result.

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

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

- Execute the decode stage for the first operation;

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

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

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

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

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

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

Peak performance

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

3.2.2 8-bit, 16-bit, 32-bit, 64-bit

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

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

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

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

3.2.3 Caches: on-chip memory

The bulk of computer memory is in chips that are separate from the processor. However, there is usually a small amount (typically a few megabytes) of on-chip memory, called the *cache*. This will be explained in detail in section 3.3.4.

3.2.4 Graphics, controllers, special purpose hardware

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

3.2.5 Superscalar processing and instruction-level parallelism

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

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

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

- multiple-issue: instructions that are independent can be started at the same time;
- pipelining: already mentioned, arithmetic units can deal with multiple operations in various stages of completion;
- branch prediction and speculative execution: a compiler can 'guess' whether a conditional instruction will evaluate to true, and execute those instructions accordingly;
- out-of-order execution: instructions can be rearranged if they are not dependent on each other, and if the resulting execution will be more efficient;
- *prefetching*: data can be speculatively requested before any instruction needing it is actually encountered (this is discussed further in section 3.3.5).

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

As clock frequency has gone up, the processor pipeline has grown in length to make the segments executable in less time. You have already seen that longer pipelines have a larger $n_{1/2}$, so more independent instructions are needed to make the pipeline run at full efficiency. As the limits to instruction-level parallelism are reached, making pipelines longer (sometimes called 'deeper') no longer pays off. This is generally seen as the reason that chip designers have moved to *multicore* architectures as a way of more efficiently using the transistors on a chip; section 3.4.

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

3.3 Memory Hierarchies

We will now refine the picture of the Von Neuman architecture, in which data is loaded immediately from memory to the processors, where it is operated on. This picture is unrealistic because of the so-called *memory wall* [38]: the

memory is too slow to load data into the process at the rate the processor can absorb it. Specifically, a single load can take 1000 cycles, while a processor can perform several operations per cycle. (After this long wait for a load, the next load can come faster, but still too slow for the processor. This matter of wait time versus throughput will be addressed below in section 3.3.2.)

In reality, there will be various memory levels in between the FPU and the main memory: the register and the caches, together called the *memory hierarchy*. These try to alleviate the memory wall problem by making recently used data available quicker than it would be from main memory. Of course, this presupposes that the algorithm and its implementation allow for data to be used multiple times.

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

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

3.3.1 Busses

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

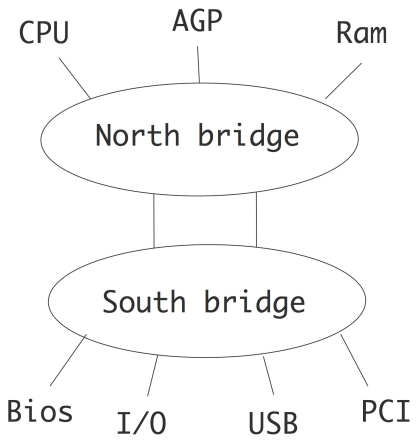


Figure 3.3 Bus structure of a processor

The bus is typically slower than the processor, operating with clock frequencies slightly in excess of 1GHz, which is a fraction of the CPU clock frequency. This is one reason that caches are needed; the fact that a processors can consume many data items per clock tick contributes to this. Apart from the frequency, the bandwidth of a bus is also determined by the number of bits that can be moved per clock cycle. This is typically 64 or 128 in current

architectures. We will now discuss this in some more detail.

3.3.2 Latency and Bandwidth

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

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

Latency is the delay between the processor issuing a request for a memory item, and the item actually arriving. We can distinguish between various latencies, such as the transfer from memory to cache, cache to register, or summarize them all into the latency between memory and processor. Latency is measured in (nano) seconds, or clock periods.

If a processor executes instructions in the order they are found in the assembly code, then execution will often *stall* while data is being fetched from memory; this is also called *memory stall*. For this reason, a low latency is very important. In practice, many processors have 'out-of-order execution' of instructions, allowing them to perform other operations while waiting for the requested data. Programmers can take this into account, and code in a way that achieves *latency hiding*. Graphical Processing Units (GPUs) can switch very quickly between threads in order to achieve latency hiding.

Bandwidth is the rate at which data arrives at its destination, after the initial latency is overcome. Bandwidth is measured in bytes (kilobytes, megabytes, gigabytes) per second or per clock cycle. The bandwidth between two memory levels is usually the product of the cycle speed of the channel (the *bus speed*) and the *bus width*: the number of bits that can be sent simultaneously in every cycle of the bus clock.

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

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

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

Typically, the further away from the processor one gets, the longer the latency is, and the lower the bandwidth. These two factors make it important to program in such a way that, if at all possible, the processor uses data from cache or register, rather than from main memory. To illustrate that this is a serious matter, consider a vector addition

```
for (i)
  a[i] = b[i]+c[i]
```

Each iteration performs one floating point operation, which modern CPU can do in one clock cycle by using pipelines. However, each iteration needs two numbers loaded and one written, for a total of 24 bytes [5] of memory traffic. Typical memory bandwidth figures (see for instance figure 3.4) are nowhere near 24 (or 32) bytes per cycle. This means that, without caches, algorithm performance can be bounded by memory performance. Of course, caches will not speed up every operations, and in fact will have no effect on the above example.

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

3.3.3 Registers

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

```
a := b + c
```

is actually implemented as

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

Looking at assembly code (for instance the output of a compiler), you see the explicit load, compute, and store instructions.

Compute instructions such as add or multiply only operate on registers. For instance, in *assembly language* you will see instructions such as

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

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

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

In this chapter you will see stressed that moving data from memory is relatively expensive. Therefore, it would be a simple optimization to leave data in register when possible. For instance, if the above computation is followed by a statement

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

[5] Actually, `a[i]` is loaded before it can be written, so there are 4 memory access, with a total of 32 bytes, per iteration.

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

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

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

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

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

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

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

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

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

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

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

```
register double t;
```

3.3.4 Caches

In between the registers, which contain the immediate input and output data for instructions, and the main memory where lots of data can reside for a long time, are various levels of *cache* memory, that have lower latency and higher bandwidth than main memory and where data are kept for an intermediate amount of time.

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

On a historical note, the notion of levels of memory hierarchy was already discussed in 1946 [8], motivated by the slowness of the memory technology at the time.

A motivating example

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

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

The assembly code would then be

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

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

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

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

There is an important difference between cache memory and registers: while data is moved into register by explicit assembly instructions, the move from main memory to cache is entirely done by hardware. Thus cache use and reuse is outside of direct programmer control.

Cache tags

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

Cache levels, speed and size

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

Data needed in some operation gets copied into the various caches on its way to the processor. If, some instructions later, a data item is needed again, it is first searched for in the L1 cache; if it is not found there, it is searched for in the L2 cache; if it is not found there, it is loaded from main memory. Finding data in cache is called a *cache hit*, and not finding it a *cache miss*.

Figure 3.4 illustrates the basic facts of the *cache hierarchy*, in this case for the *Intel Sandy Bridge* chip: the closer caches are to the FPU, the faster, but also the smaller they are. Some points about this figure.

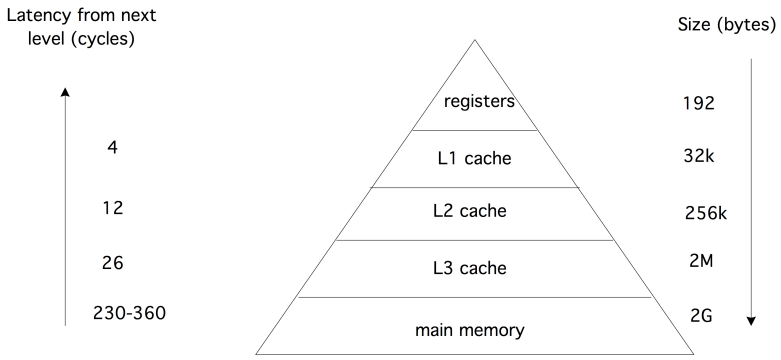


Figure 3.4 Memory hierarchy of an Intel Sandy Bridge, characterized by speed and size.

- Loading data from registers is so fast that it does not constitute a limitation on algorithm execution speed. On the other hand, there are few registers. Each core has 16 general purpose registers, and 16 SIMD registers.
- The L1 cache is small, but sustains a bandwidth of 32 bytes, that is 4 double precision number, per cycle. This is enough to load two operands each for two operations, but note that the core can actually perform 4 operations per cycle. Thus, to achieve peak speed, certain operands need to stay in register: typically, L1 bandwidth is enough for about half of peak performance.
- The bandwidth of the L2 and L3 cache is nominally the same as of L1. However, this bandwidth is partly wasted on coherence issues.
- Main memory access has a latency of more than 100 cycles, and a bandwidth of 4.5 bytes per cycle, which is about 1/7th of the L1 bandwidth. However, this bandwidth is shared by the multiple cores of a processor chip, so effectively the bandwidth is a fraction of this number. Most clusters will also have more than one *socket* (processor chip) per node, typically 2 or 4, so some bandwidth is spent on maintaining *cache coherence* (see section 3.4), again reducing the bandwidth available for each chip.

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

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

Exercise 3.1 The L1 cache is smaller than the L2 cache, and if there is an L3, the L2 is smaller than the L3. Give a practical and a theoretical reason why this is so.

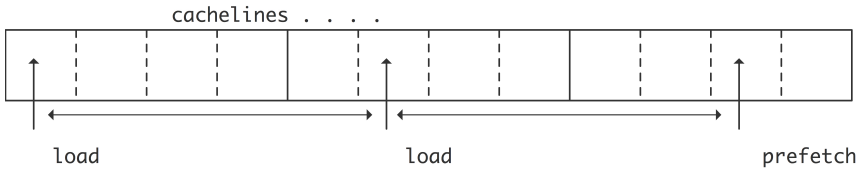


Figure 3.5 Prefetch stream generated by equally spaced requests

Cache memory versus regular memory

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

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

Loads versus stores

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

3.3.5 Prefetch streams

In the traditional von Neumann model (section 3.1), each instruction contains the location of its operands, so a CPU implementing this model would make a separate request for each new operand. In practice, often subsequent data items are adjacent or regularly spaced in memory. The memory system can try to detect such data patterns by looking at cache miss points, and request a *prefetch data stream*; figure 3.5.

In its simplest form, the CPU will detect that consecutive loads come from two consecutive cache lines, and automatically issue a request for the next following cache line. This process can be repeated or extended if the code makes an actual request for that third cache line. Since these cache lines are now brought from memory well before they are needed, prefetch has the possibility of eliminating the latency for all but the first couple of data items.

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

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

Introducing prefetch by the programmer is a careful balance of a number of factors [24]. Prime among these is the *prefetch distance*: the number of cycles between the start of the prefetch and when the data is needed. In practice, this is often the number of iterations of a loop: the prefetch instruction requests data for a future iteration.

3.3.6 Concurrency and memory transfer

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

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

For full utilization of the bandwidth, at all times a volume of data equal to the bandwidth times the latency has to be in flight. Since these data have to be independent, we get a statement of *Little's law* [33]:

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

This is illustrated in figure 3.6. The problem with maintaining this concurrency

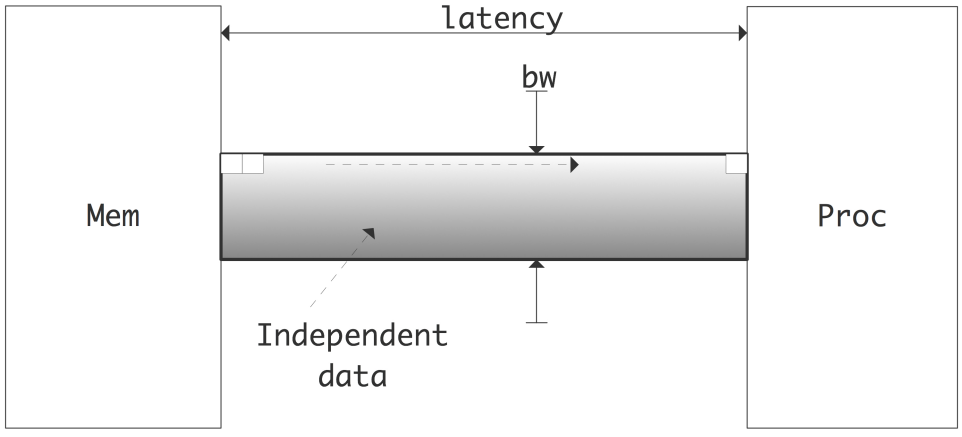


Figure 3.6 Illustration of Little's Law that states how much independent data needs to be in flight

is not that a program does not have it; rather, the program is to get the compiler and runtime system recognize it. For instance, if a loop traverses a long array, the compiler will not issue a large number of memory requests. The prefetch mechanism (section 3.3.5) will issue some memory requests ahead of time, but

typically not enough. Thus, in order to use the available bandwidth, multiple streams of data need to be under way simultaneously. Therefore, we can also phrase Little's law as

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

3.4 Multicore architectures

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

- Clock frequency can not be increased further, since it increases energy consumption, heating the chips too much; see section 3.6.1.
- It is not possible to extract more Instruction Level Parallelism (ILP) from codes, either because of compiler limitations, because of the limited amount of intrinsically available parallelism, or because branch prediction makes it impossible (see section 3.2.5).

One of the ways of getting a higher utilization out of a single processor chip is then to move from a strategy of further sophistication of the single processor, to a division of the chip into multiple processing 'cores' [6]. The separate cores can work on unrelated tasks, or, by introducing what is in effect data parallelism (section 4.3.1), collaborate on a common task at a higher overall efficiency[34].

This solves the above two problems:

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

While the first multicore CPU were simply two processors on the same die, later generations incorporated L3 or L2 caches that were shared between the two processor cores; see figure 3.7 This design makes it efficient for the cores to work jointly on the same problem. The cores would still have their own L1 cache, and these separate caches lead to a *cache coherence* problem.

We note that the term 'processor' is now ambiguous: it can refer to either the chip, or the processor core on the chip. For this reason, we mostly talk about a *socket* for the whole chip and *core* for the part containing one arithmetic and logic unit and having its own registers. Currently, CPU with 4 or 6 cores are common, even in laptops, and Intel and AMD are marketing 12-core chips. The core count is likely to go up in the future: Intel has already shown an 80-core prototype that is developed into the 48 core 'Single-chip Cloud Computer', illustrated in fig 3.8. This chip has a structure with 24 dual-core 'tiles' that are connected through a 2D mesh network. Only certain tiles are connected

[6] Another solution is Intel's *hyperthreading*, which lets a processor mix the instructions of several instruction streams. The benefits of this are strongly dependent on the individual case. However, this same mechanism is exploited with great success in GPUs. For a discussion see section 5.1.8

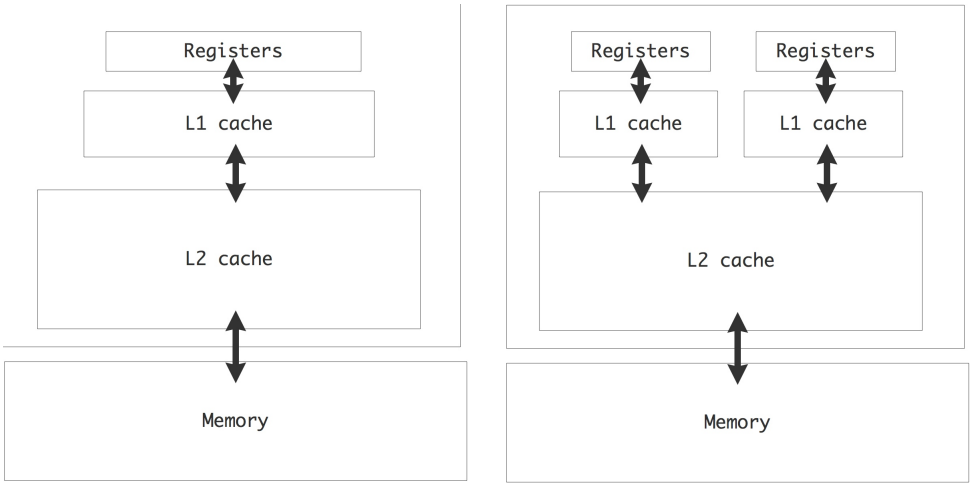


Figure 3.7 Cache hierarchy in a single-core and dual-core chip

to a memory controller, others can not reach memory other than through the on-chip network.

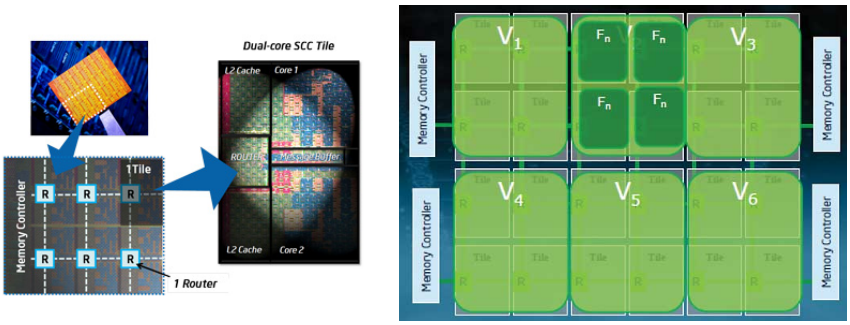


Figure 3.8 Structure of the Intel Single-chip Cloud Computer chip

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

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

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

Node There can be multiple sockets on a single 'node' or motherboard, accessing the same shared memory.

Network Distributed memory programming (see the next chapter) is needed to let nodes communicate.

Historically, multicore architectures have a precedent in multiprocessor shared memory designs (section 4.4.1) such as the *Sequent Symmetry* and the *Alliant FX/8*. Conceptually the program model is the same, but the technology now allows to shrink a multiprocessor board to a multicore chip.

3.4.1 Computations on multicore chips

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

The MPI library (section 5.3.3) is typically used to communicate between processors that are connected through a network. However, it can also be used in a single multicore processor: the MPI calls then are realized through shared memory copies.

Alternatively, we can use the shared memory and shared caches and program using threaded systems such as OpenMP (section 5.2). The advantage of this mode is that parallelism can be much more dynamic, since the runtime system can set and change the correspondence between threads and cores during the program run.

3.5 Node architecture and sockets

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

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

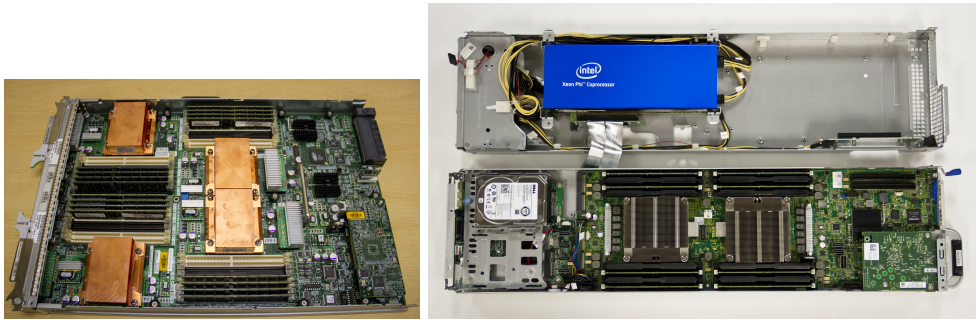


Figure 3.9 Left: a four-socket design. Right: a two-socket design with co-processor.

spread over banks that are directly attached to one particular socket. This is for instance illustrated in figure 3.9, which shows the four-socket node of the *Ranger* supercomputer (no longer in production) and the two-socket node of the *Stampede* supercomputer which contains an *Intel Xeon Phi* co-processor. In both designs you clearly see the memory chips that are directly connected to the sockets.

This is an example of *Non-Uniform Memory Access* (NUMA) design: for a process running on some core, the memory attached to its socket is slightly faster to access than the memory attached to another socket.

One result of this is the *first-touch* phenomenon. Dynamically allocated memory is not actually allocated until it's first written to. Consider now the following OpenMP (section 5.2) code:

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

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

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

3.6 Further topics

3.6.1 Power consumption

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

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

Derivation of scaling properties

The properties of *constant field scaling* or *Dennard scaling* [5], [13] are an ideal-case description of the properties of a circuit as it is miniaturized. One important result is that power density stays constant as chip features get smaller, and the frequency is simultaneously increased.

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

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

Then we can derive that

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

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

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

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

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

(3.1)

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

Multicore

At the time of this writing (circa 2010), miniaturization of components has almost come to a standstill, because further lowering of the voltage would give prohibitive leakage. Conversely, the frequency can not be scaled up since this would raise the heat production of the chip too far. Figure 3.10 gives a dramatic

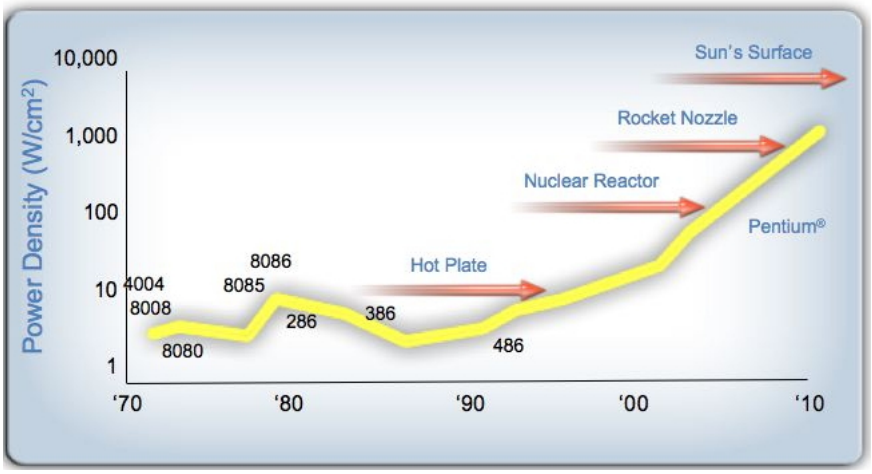


Figure 3.10 Projected heat dissipation of a CPU if trends had continued – this graph courtesy Pat Helsinger

illustration of the heat that a chip would give off, if single-processor trends had continued.

One conclusion is that computer design is running into a *power wall*, where the sophistication of a single core can not be increased any further (so we can for instance no longer increase *ILP* and *pipeline depth*) and the only way to increase performance is to increase the amount of explicitly visible parallelism. This development has led to the current generation of *multicore* processors; see section 3.4. It is also the reason GPUs with their simplified processor design and hence lower energy consumption are attractive; the same holds for FPGAs. One solution to the power wall problem is introduction of *multicore* processors. Recall equation 3.1, and compare a single processor to two processors at half the frequency. That should have the same computing power, right? Since we

lowered the frequency, we can lower the voltage if we stay with the same process technology.

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

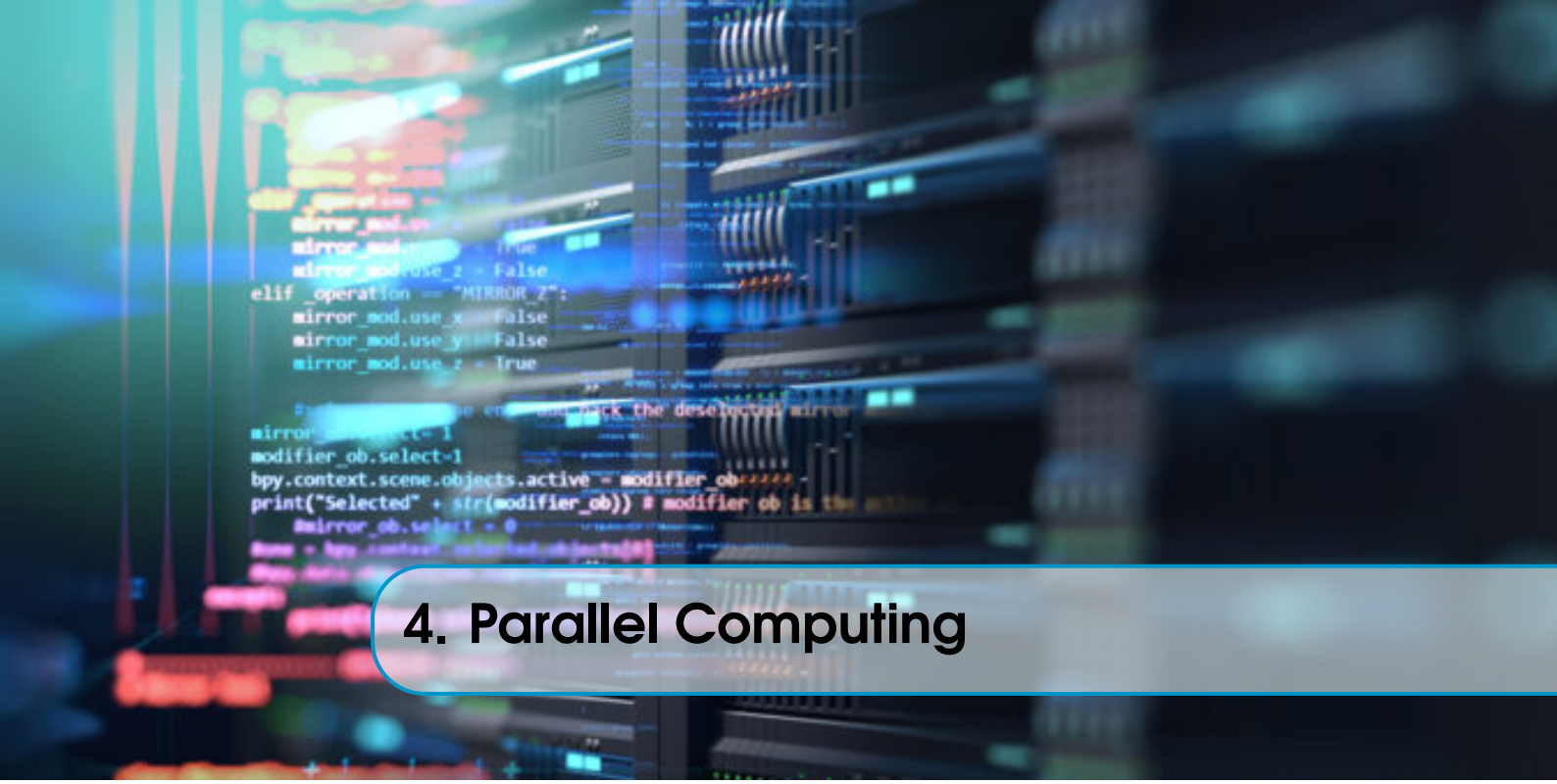
$$\left.\begin{array}{l} C_{\text{multi}} = 2C \\ F_{\text{multi}} = F/2 \\ V_{\text{multi}} = V/2 \end{array}\right\} \Rightarrow P_{\text{multi}} = P/4.$$

In practice the capacitance will go up by a little over 2, and the voltage can not quite be dropped by 2, so it is more likely that $P_{\text{multi}} \approx 0.4 \times P$ [9]. Of course the integration aspects are a little more complicated in practice [6]; the important conclusion is that now, in order to lower the power (or, conversely, to allow further increase in performance while keeping the power constant) we now have to start programming in parallel.

Total computer power

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

In the most recent generations of processors, power is becoming an overriding consideration, with influence in unlikely places. For instance, the Single Instruction Multiple Data (SIMD) design of processors (see section 4.3.1, in particular section 4.3.1) is dictated by the power cost of instruction decoding.



4. Parallel Computing

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

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

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

4.1 Introduction

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

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

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

can be done with up to n processors. In the idealized case with n processors, each processor has local scalars `a,b,c` and executes the single instruction `a=b+c`. This is depicted in figure 4.1.

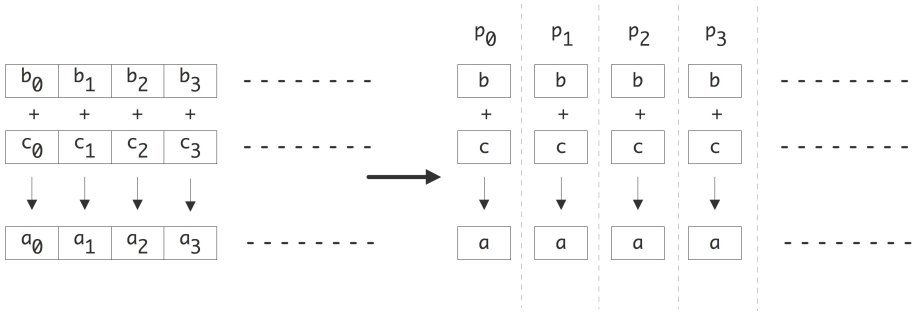


Figure 4.1 Parallelization of a vector addition

In the general case, where each processor executes something like

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

execution time is linearly reduced with the number of processors. If each operation takes a unit time, the original algorithm takes time n, and the parallel execution on p processors n/p. The parallel algorithm is faster by a factor of p [7].

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

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

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

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

there is a way to parallelize it: every iteration of the outer loop is now a loop that can be done by n/s processors in parallel. Since the outer loop will go through $\log_2 n$ iterations, we see that the new algorithm has a reduced runtime of $n/p \cdot \log_2 n$. The parallel algorithm is now faster by a factor of $p/\log_2 n$. This is depicted in figure 4.2.

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

- Sometimes algorithms need to be rewritten slightly to make them parallel.

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

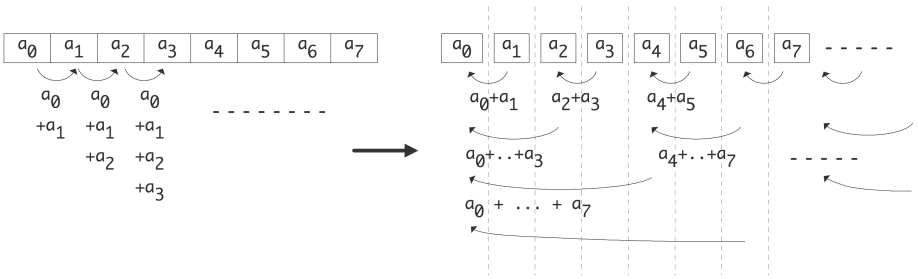


Figure 4.2 Parallelization of a vector reduction

- A parallel algorithm may not show perfect speedup.

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

First let us look systematically at communication. We can take the parallel algorithm in the right half of figure 4.2 and turn it into a tree graph by defining the inputs as leaf nodes, all partial sums as interior nodes, and the root as the total sum. There is an edge from one node to another if the first is input to the (partial) sum in the other. This is illustrated in figure 4.3. In this figure nodes are horizontally aligned with other computations that can be performed simultaneously; each level is sometimes called a *superstep* in the computation. Nodes are vertically aligned if they are computed on the same processors, and an arrow corresponds to a communication if it goes from one processor to another. The vertical alignment in figure 4.3 is not the only one

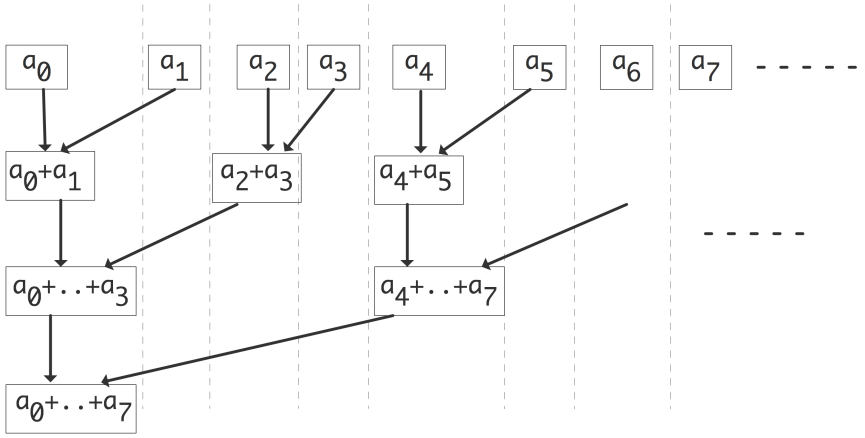


Figure 4.3 Communication structure of a parallel vector reduction

possible. If nodes are shuffled within a superstep or horizontal level, a different communication pattern arises.

Exercise 4.1 Consider placing the nodes within a superstep on random processors. Show that, if no two nodes wind up on the same processor, at most twice the number of communications is performed from the case in figure 4.3.

Processors are often connected through a network, and moving data through this network takes time. This introduces a concept of distance between the processors. This is easily seen in figure 4.3 where the processors are linearly ordered. If the network only connects a processor with its immediate neighbours, each iteration of the outer loop increases the distance over which communication takes place.

Exercise 4.2 Assume that an addition takes a certain unit time, and that moving a number from one processor to another takes that same unit time. Show that the communication time equals the computation time.

Now assume that sending a number from processor p to $p \pm k$ takes time k. Show that the execution time of the parallel algorithm now is of the same order as the sequential time.

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

Exercise 4.3 Consider the case of summing 8 elements with 4 processors. Show that some of the edges in the graph of figure 4.3 no longer correspond to actual communications. Now consider summing 16 elements with, again, 4 processors. What is the number of communication edges this time?

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

4.1.1 Functional parallelism versus data parallelism

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

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

It is also possible to find independence, not based on data elements, but based on the instructions themselves. Traditionally, compilers analyze code in terms of ILP: independent instructions can be given to separate floating point units, or reordered, for instance to optimize register usage (see also section 4.5.2). ILP is one case of *functional parallelism*; on a higher level, functional parallelism can be obtained by considering independent subprograms, often called *task parallelism*; see section 4.5.3.

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

4.1.2 Parallelism in the algorithm versus in the code

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

Exercise 4.4

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

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

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

4.2 Theoretical concepts

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

4.2.1 Definitions

Speedup and efficiency

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

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

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

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

Exercise 4.5 The case of speedup larger than the number of processors is called *superlinear speedup*. Give a theoretical argument why this can never happen.

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

Cost-optimality

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

$$T_o = pT_p - T1.$$

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

We will later see two different types of overhead:

1. The parallel algorithm can be essentially different from the sequential one. For instance, sorting algorithms have a complexity $O(n \log n)$, but the parallel bitonic sort has complexity $O(n \log^2 n)$.
2. The parallel algorithm can have overhead derived from the process or parallelizing, such as the cost of sending messages.

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

Exercise 4.6 The definition of overhead above implicitly assumes that overhead is not parallelizable. Discuss this assumption in the context of the two examples above.

Critical path

The above definitions of speedup and efficiency made an implicit assumption that parallel work can be arbitrarily subdivided. As you saw in the summing example in section 4.1, this may not always be the case: there can be dependencies between operations, which limits the amount of parallelism that can be employed.

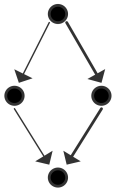
We define the *critical path* as a (possibly non-unique) chain of dependencies of maximum length. Since the tasks on a critical path need to be executed one after another, the length of the critical path is a lower bound on parallel execution time.

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

Definition 4.1
T_1 : the time the computation takes on a single processor
T_p : the time the computation takes with p processors
T_∞ : the time the computation takes with unlimited processors available
P_∞ : the value of p for which $T_p = T_\infty$

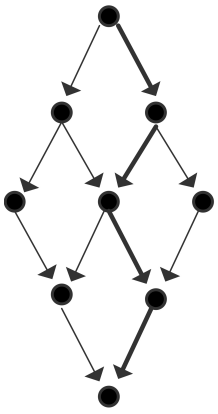
With these concepts, we can define the *average parallelism* of an algorithm as T_1/T_∞, and the length of the critical path is T_∞.

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



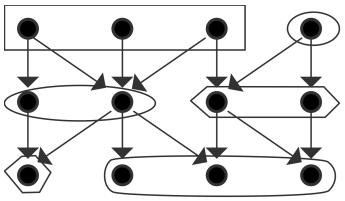
The maximum number of processors that can be used is 2 and the average parallelism is 4/3:

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



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

$$T_1 = 9, \quad T_\infty = 5 \quad \Rightarrow T_1/T_\infty = 9/5$$
$$T_2 = 6, \quad S_2 = 3/2, \quad E_2 = 3/4$$
$$T_3 = 5, \quad S_3 = 9/5, \quad E_3 = 3/5$$
$$P_\infty = 3$$



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

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

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

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

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

Theorem 4.2.1 — Brent's Theorem. Let m be the total number of tasks, p the number of processors, and t the length of a *critical path*. Then the computation can be done in

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

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

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

■

4.2.2 Asymptotics

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

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

Exercise 4.7 Show that this algorithm ignores some serious issues about memory usage:

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

Adding N numbers $\{x_i\}_{i=1\ldots N}$ can be performed in $\log_2 N$ time with $N/2$ processors. As a simple example, consider the sum of n numbers: $s = \sum_{i=1}^{n} a_i$. If we have $n/2$ processors we could compute:

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

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

It is now a legitimate theoretical question to ask

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

Such questions have been researched (see for instance [25]), but they have little bearing on high performance computing.

A first objection to these kinds of theoretical bounds is that they implicitly assume some form of shared memory. In fact, the formal model for these algorithms is called a *Parallel Random Access Memory* (PRAM), where the assumption is that every memory location is accessible to any processor. Often an additional assumption is made that multiple simultaneous accesses to the same location are in fact possible[8]. These assumptions are unrealistic in practice, especially in the context of scaling up the problem size and the number of processors. A further objection to the PRAM model is that even on a single processor it ignores the memory hierarchy; section 3.3.

But even if we take distributed memory into account, theoretical results can still be unrealistic. The above summation algorithm can indeed work unchanged in distributed memory, except that we have to worry about the distance between active processors increasing as we iterate further. If the processors are connected by a linear array, the number of 'hops' between active

[8] This notion can be made precise; for instance, one talks of a CREW-PRAM, for Concurrent Read, Exclusive Write PRAM.

processors doubles, and with that, asymptotically, the computation time of the iteration. The total execution time then becomes $n/2$, a disappointing result given that we throw so many processors at the problem.

Finally, the question 'what if we had infinitely many processors' is not realistic as such, but we will allow it in the sense that we will ask the *weak scaling* question (section 4.2.4) 'what if we let the problem size and the number of processors grow proportional to each other'. This question is legitimate, since it corresponds to the very practical deliberation whether buying more processors will allow one to run larger problems, and if so, with what 'bang for the buck'.

4.2.3 Amdahl's law

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

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

$$T_P = T_1(F_s + F_p/P). \tag{4.1}$$

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

The sequential fraction of a code can consist of things such as I/O operations. However, there are also parts of a code that in effect act as sequential. Consider a program that executes a single loop, where all iterations can be computed independently. Clearly, this code is easily parallelized. However, by splitting the loop in a number of parts, one per processor, each processor now has to deal with loop overhead: calculation of bounds, and the test for completion. This overhead is replicated as many times as there are processors. In effect, loop overhead acts as a sequential part of the code.

Exercise 4.8 Let's do a specific example. Assume that a code has a setup that takes 1 second and a parallelizable section that takes 1000 seconds on one processor. What are the speedup and efficiency if the code is executed with 100 processors? What are they for 500 processors? Express your answer to at most two significant digits.

Exercise 4.9 Investigate the implications of Amdahl's law: if the number of processors P increases, how does the parallel fraction of a code have to increase to maintain a fixed efficiency?

Amdahl's law with communication overhead

In a way, Amdahl's law, sobering as it is, is even optimistic. Parallelizing a code will give a certain speedup, but it also introduces *communication overhead* that will lower the speedup attained. Let us refine our model of equation (4.1) (see [32, p. 367]):

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

where T_c is a fixed communication time.

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

$$S_p = \frac{T_1}{T_1/p + T_c}. \tag{4.2}$$

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

Gustafson's law

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

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

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

Now we have two possible definitions of T_1. First of all, there is the T_1 you get from setting $p = 1$ in T_p. (Convince yourself that that is actually the same as T_p.) However, what we need is T_1 describing the time to do all the operations of the parallel program. This is:

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

This gives us a speedup of

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

That is, speedup is now a function that decreases from p, linearly with p.

As with Amdahl's law, we can investigate the behaviour of Gustafson's law if we include communication overhead. Let's go back to equation (4.2) for a perfectly parallelizable problem, and approximate it as

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

Now, under the assumption of a problem that is gradually being scaled up, T_c, T_1 become functions of p. We see that if $T_1(p) \sim pT_c(p)$, we get linear speedup that is a constant fraction away from 1. In general we can not take this analysis further.

4.2.4 Scalability

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

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

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

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

More interestingly, *weak scalability* is a more vaguely defined term. It describes the behaviour of execution, as problem size and number of processors both grow, but in such a way that the amount of data per processor stays constant. Measures such as speedup are somewhat hard to report, since the relation between the number of operations and the amount of data can be complicated. If this relation is linear, one could state that the amount of data per processor is kept constant, and report that parallel execution time is constant as the number of processors grows. (Can you think of applications where the relation between work and data is linear? Where it is not?)

In terms of efficiency:

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

Exercise 4.10 We can formulate strong scaling as a runtime that is inversely proportional to the number of processors:

$$t = c/p.$$

Show that on a log-log plot, that is, you plot the logarithm of the runtime against the logarithm of the number of processors, you will get a straight line with slope -1.

Can you suggest a way of dealing with a non-parallelizable section, that is, with a runtime $t = c_1 + c_2/p$?

Exercise 4.11 Suppose you are investigating the weak scalability of a code. After running it for a couple of sizes and corresponding numbers of processors, you find that in each case the flop rate is roughly the same. Argue that the code is indeed weakly scalable.

Iso-efficiency

In the definition of *weak*scalability above, we stated that, under some relation between problem size N and number of processors P, efficiency will stay constant. We can make this precise and define the *iso-efficiency curve* as the relation between N, P that gives constant efficiency [19].

4.2.5 Concurreny; asynchronous and distributed computing

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

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

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

We will not discuss this topic much in this book. Section 5.1 discusses the *thread* mechanism that supports time slicing; on modern multicore processors threads can be used to implement shared memory parallel computing.

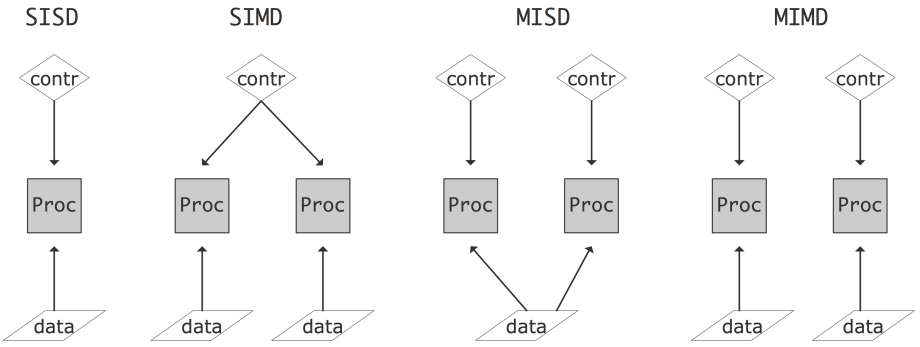


Figure 4.4 The four classes of the Flynn's taxonomy

The book 'Communicating Sequential Processes' offers an analysis of the interaction between concurrent processes [28]. Other authors use topology to analyze asynchronous computing [27].

4.3 Parallel Computers Architectures

For quite a while now, the top computers have been some sort of parallel computer, that is, an architecture that allows the simultaneous execution of multiple instructions or instruction sequences. One way of characterizing the various forms this can take is due to Flynn [18]. *Flynn's taxonomy* characterizes architectures by whether the *data flow* and *control flow* are shared or independent. The following four types result (see also figure 4.4):

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

SIMD Single Instruction Multiple Data: in this computer type there can be multiple processors, each operating on its own data item, but they are all executing the same instruction on that data item.

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

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

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

4.3.1 SIMD

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

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

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

On the other hand, there are operations that can not can be executed efficiently on an array processor. For instance, evaluating a number of terms of a recurrence $x_{i+1} = ax_i + b_i$ involves that many additions and multiplications, but they alternate, so only one operation of each type can be processed at any one time. There are no arrays of numbers here that are simultaneously the input of an addition or multiplication.

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

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

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

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

True SIMD in CPUs and GPUs

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

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

The AVX instructions are based on up to 512-bit wide SIMD, that is, eight floating point numbers can be processed simultaneously. Just as single floating point operations operate on data in registers (section 3.3.3), vector operations use *vector registers*. The locations in a vector register are sometimes referred to as *SIMD lanes*.

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

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

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

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

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

4.3.2 MIMD / SPMD computers

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

There is a great variety in MIMD computers. Some of the aspects concern the way memory is organized, and the network that connects the processors. Apart from these hardware aspects, there are also differing ways of programming these machines. We will see all these aspects below. Many machines these days are called *clusters*. They can be built out of custom or commodity processors (if they consist of PCs, running Linux, and connected with *Ethernet*, they are referred to as *Beowulf clusters* [23]); since the processors are independent they are examples of the MIMD or SPMD model.

4.4 Different types of memory access

In the introduction we defined a parallel computer as a setup where multiple processors work together on the same problem. In all but the simplest cases this means that these processors need access to a joint pool of data. In the previous chapter you saw how, even on a single processor, memory can have a

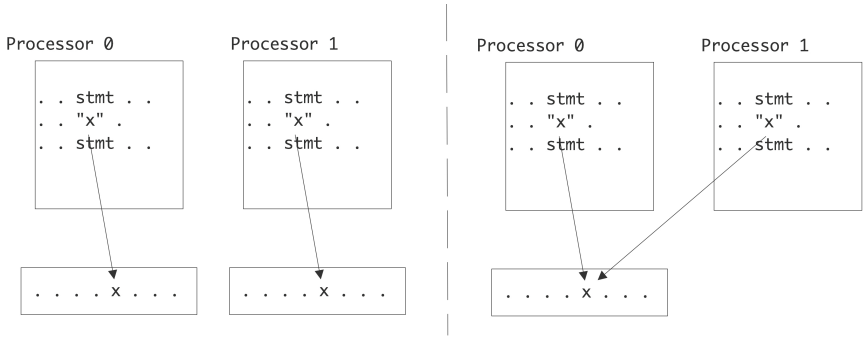


Figure 4.5 References to identically named variables in the distributed and shared memory case

hard time keeping up with processor demands. For parallel machines, where potentially several processors want to access the same memory location, this problem becomes even worse. We can characterize parallel machines by the approach they take to the problem of reconciling multiple accesses, by multiple processes, to a joint pool of data.

The main distinction here is between *distributed memory* and *shared memory*. With distributed memory, each processor has its own physical memory, and more importantly its own *address space*. Thus, if two processors refer to a variable `x`, they access a variable in their own local memory. On the other hand, with shared memory, all processors access the same memory; we also say that they have a *shared address space*. Thus, if two processors both refer to a variable `x`, they access the same memory location.

4.4.1 Symmetric Multi-Processors: Uniform Memory Access

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

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

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

4.4.2 Non-Uniform Memory Access

The UMA approach based on shared memory is obviously limited to a small number of processors. The crossbar networks are expandable, so they would seem the best choice. However, in practice one puts processors with a local

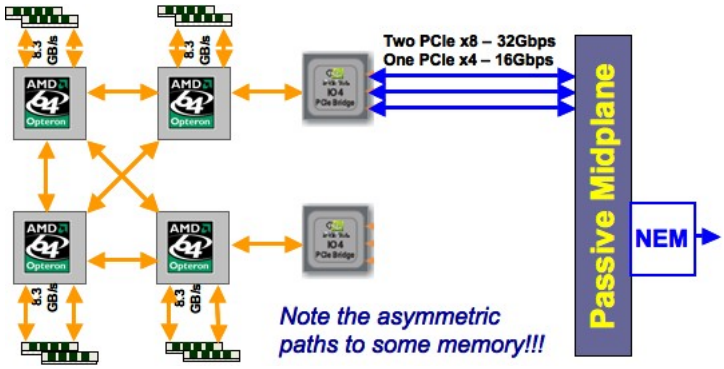


Figure 4.6 Non-uniform memory access in a four-socket motherboard

memory in a configuration with an exchange network. This leads to a situation where a processor can access its own memory fast, and other processors' memory slower. This is one case of so-called *Non-Uniform Memory Access (NUMA)*: a strategy that uses physically distributed memory, abandoning the uniform access time, but maintaining the logically shared address space: each processor can still access any memory location.

Figure 4.6 illustrates NUMA in the case of the four-socket motherboard of the Ranger supercomputer. Each chip has its own memory (8Gb) but the motherboard acts as if the processors have access to a shared pool of 32Gb. Obviously, accessing the memory of another processor is slower than accessing local memory. In addition, note that each processor has three connections that could be used to access other memory, but the rightmost two chips use one connection to connect to the network. This means that accessing each other's memory can only happen through an intermediate processor, slowing down the transfer, and tying up that processor's connections.

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

Keeping copies of a memory location synchronized is known as *cache coherence*; a multi-processor system using it is sometimes called a 'cache-coherent NUMA' or *ccNUMA* architecture.

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

Among current vendors only SGI (the *UV* line) and Cray (the *XE6*) market products with large scale NUMA. Both offer strong support for *Partitioned Global Address Space (PGAS)* languages. There are vendors, such as *ScaleMP*,

that offer a software solution to distributed shared memory on regular clusters.

4.4.3 Logically and physically distributed memory

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

With logically *and* physically distributed memory, the only way one processor can exchange information with another is through passing information explicitly through the network. You will see more about this in section 5.3.3.

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

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

4.5 Granularity of parallelism

Let us take a look at the question 'how much parallelism is there in a program execution'. There is the theoretical question of the absolutely maximum number of actions that can be taken in parallel, but we also need to wonder what kind of actions these are and how hard it is to actually execute them in parallel, as well has how efficient the resulting execution is.

The discussion in this section will be mostly on a conceptual level; in section 5 we will go into some detail on how parallelism can actually be programmed.

4.5.1 Data parallelism

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

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

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

```
a = 2*b
```

on its local data.

If your code consists predominantly of such loops over arrays, it can be executed efficiently with all processors in lockstep. Architectures based on this idea, where the processors can in fact *only* work in lockstep, have existed, see section 4.3.1. Such fully parallel operations on arrays appear in computer graphics, where every pixel of an image is processed independently. For this reason, GPUs are strongly based on data parallelism.

4.5.2 Instruction-level parallelism

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

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

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

4.5.3 Task-level parallelism

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

if optimal(*root*) **then**
  exit

else
  parallel : mathbfSearchInTree (leftchild), **SearchInTree** (rightchild)
  Procedure SearchInTree(root)

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

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

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

Unlike in the data parallel example above, the assignment of data to processor is not determined in advance in such a scheme. Therefore, this mode of parallelism is most suited for thread-programming, for instance through the OpenMP library; section 5.2.

4.5.4 Conveniently parallel computing

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

4.5.5 Medium-grain data parallelism

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

```
my_lower_bound = // some processor-dependent number
my_upper_bound = // some processor-dependent number
for (i=my_lower_bound; i<my_upper_bound; i++)
  // the loop body goes here
```

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

4.6 Topologies

If a number of processors are working together on a single task, most likely they need to communicate data. For this reason there needs to be a way for data to make it from any processor to any other. In this section we will discuss some of the possible schemes to connect the processors in a parallel machine. Such a scheme is called a (processor) *topology*.

In order to get an appreciation for the fact that there is a genuine problem here, consider two simple schemes that do not 'scale up':

- *Ethernet* is a connection scheme where all machines on a network are on a single cable (see remark below). If one machine puts a signal on the wire to send a message, and another also wants to send a message, the latter will detect that the sole available communication channel is occupied, and it will wait some time before retrying its send operation. Receiving data on ethernet is simple: messages contain the address of the intended recipient, so a processor only has to check whether the signal on the wire is intended for it.

The problems with this scheme should be clear. The capacity of the communication channel is finite, so as more processors are connected to it, the capacity available to each will go down. Because of the scheme for resolving conflicts, the average delay before a message can be started will also increase.

- In a *fully connected* configuration, each processor has one wire for the communications with each other processor. This scheme is perfect in the sense that messages can be sent in the minimum amount of time, and two messages will never interfere with each other. The amount of data that can be sent from one processor is no longer a decreasing function of the number of processors; it is in fact an increasing function, and if the network controller can handle it, a processor can even engage in multiple simultaneous communications.
  The problem with this scheme is of course that the design of the network interface of a processor is no longer fixed: as more processors are added to the parallel machine, the network interface gets more connecting wires. The network controller similarly becomes more complicated, and the cost of the machine increases faster than linearly in the number of processors.

The above description of Ethernet is of the original design. With the use of switches, especially in an HPC context, this description does not really apply anymore.

It was initially thought that ethernet would be inferior to other solutions such as IBM's *token ring* network. It takes fairly sophisticated statistical analysis to prove that it works a lot better than was naively expected.

4.6.1 Bandwidth and latency

The assumption that sending a message can be considered a unit time operation, is of course unrealistic. A large message will take longer to transmit than a short one. There are two concepts to arrive at a more realistic description of the transmission process; we have already seen this in section 3.3.2 in the context of transferring data between cache levels of a processor.

latency Setting up a communication between two processors takes an amount of time that is independent of the message size. The time that this takes is known as the *latency* of a message. There are various causes for this delay.

- The two processors engage in 'hand-shaking', to make sure that the recipient is ready, and that appropriate buffer space is available for receiving the message.
- The message needs to be encoded for transmission by the sender, and decoded by the receiver.
- The actual transmission may take time: parallel computers are often big enough that, even at lightspeed, the first byte of a message can take hundreds of cycles to traverse the distance between two processors.

bandwidth After a transmission between two processors has been initiated, the main number of interest is the number of bytes per second that can go through the channel. This is known as the *bandwidth*. The bandwidth can usually be determined by the *channel rate*, the rate at which a physical link can deliver bits, and the *channel width*, the number of physical wires in a link. The channel width is typically a multiple of 16, usually 64 or 128. This is also expressed by saying that a channel can send one or two 8-byte words simultaneously.

Bandwidth and latency are formalized in the expression

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

for the transmission time of an n-byte message. Here, α is the latency and β is the time per byte, that is, the inverse of bandwidth. Sometimes we consider data transfers that involve communication, for instance in the case of a *collective operation*. We then extend the transmission time formula to

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

where γ is the time per operation, that is, the inverse of the *computation rate*.

It would also be possible to refine this formulas as

$$T(n, p) = \alpha + \beta n + \delta p$$

where p is the number of network 'hops' that is traversed. However, on most networks the value of δ is far lower than of α, so we will ignore it here. Also, in fat-tree networks the number of hops is of the order of $\log P$, where P is the total number of processors, so it can never be very large anyway.

4.7 Load balancing

In much of this chapter, we assumed that a problem could be perfectly divided over processors, that is, a processor would always be performing useful work, and only be *idle* because of latency in communication. In practice, however, a processor may be idle because it is waiting for a message, and the sending processor has not even reached the send instruction in its code. Such a situation, where one processor is working and another is idle, is described as *load unbalance*: there is no intrinsic reason for the one processor to be idle, and it could have been working if we had distributed the work load differently.

There is an asymmetry between processors having too much work and having not enough work: it is better to have one processor that finishes a task early, than having one that is overloaded so that all others wait for it.

Exercise 4.12 Make this notion precise. Suppose a parallel task takes time 1 on all processors but one.

- Let $0 < \alpha < 1$ and let one processor take time $1 + \alpha$. What is the speedup and efficiency as function of the number of processors? Consider this both in the Amdahl and Gustafsson sense (section 4.2.3).

- Answer the same questions if one processor takes time $1 - \alpha$.

Load balancing is often expensive since it requires moving relatively large amounts of data. For instance, the data exchanges during for a sparse matrix-vector product is of a lower order than what is stored on the processor. However, we will not go into the actual cost of moving: our main concerns here are to balance the workload, and to preserve any locality in the original load distribution.

4.8 Remaining topics

4.8.1 Distributed computing, grid computing, cloud computing

In this section we will take a short look at terms such as *cloud computing*, and an earlier term *distributed computing*. These are concepts that have a relation to parallel computing in the scientific sense, but that differ in certain fundamental ways.

Distributed computing can be traced back as coming from large database servers, such as airline reservations systems, which had to be accessed by many travel agents simultaneously. For a large enough volume of database accesses a single server will not suffice, so the mechanism of *remote procedure call* was invented, where the central server would call code (the procedure in question) on a different (remote) machine. The remote call could involve transfer of data, the data could be already on the remote machine, or there would be some mechanism that data on the two machines would stay synchronized. This gave rise to the *Storage Area Network (SAN)*. A generation later than distributed database systems, web servers had to deal with the same problem of many simultaneous accesses to what had to act like a single server.

We already see one big difference between distributed computing and high performance parallel computing. Scientific computing needs parallelism because a single simulation becomes too big or slow for one machine; the business applications sketched above deal with many users executing small programs (that is, database or web queries) against a large data set. For scientific needs, the processors of a parallel machine (the nodes in a cluster) have to have a very fast connection to each other; for business needs no such network is needed, as long as the central dataset stays coherent.

Both in *HPC* and in business computing, the server has to stay available and operative, but in distributed computing there is considerably more liberty in how to realize this. For a user connecting to a service such as a database, it does not matter what actual server executes their request. Therefore, distributed computing can make use of *virtualization*: a virtual server can be spawned off on any piece of hardware.

An analogy can be made between remote servers, which supply computing power wherever it is needed, and the electric grid, which supplies electric power wherever it is needed. This has led to *grid computing* or *utility computing*, with the Teragrid, owned by the US National Science Foundation, as an example. Grid computing was originally intended as a way of hooking up computers

connected by a *Local Area Network (LAN)* or *Wide Area Network (WAN)*, often the Internet. The machines could be parallel themselves, and were often owned by different institutions. More recently, it has been viewed as a way of sharing resources, both datasets, software resources, and scientific instruments, over the network.

The notion of utility computing as a way of making services available, which you recognize from the above description of distributed computing, went mainstream with Google's search engine, which indexes the whole of the Internet. Another example is the GPS capability of Android mobile phones, which combines GIS, GPS, and mashup data. The computing model by which Google's gathers and processes data has been formalized in MapReduce [12]. It combines a data parallel aspect (the 'map' part) and a central accumulation part ('reduce'). Neither involves the tightly coupled neighbour-to-neighbour communication that is common in scientific computing. An open source framework for MapReduce computing exists in Hadoop [4]. Amazon offers a commercial Hadoop service.

The concept of having a remote computer serve user needs is attractive even if no large datasets are involved, since it absolves the user from the need of maintaining software on their local machine. Thus, Google Docs offers various 'office' applications without the user actually installing any software. This idea is sometimes called *Software As a Service (SAS or SaaS)*, where the user connects to an 'application server', and accesses it through a client such as a web browser. In the case of Google Docs, there is no longer a large central dataset, but each user interacts with their own data, maintained on Google's servers. This of course has the large advantage that the data is available from anywhere the user has access to a web browser.

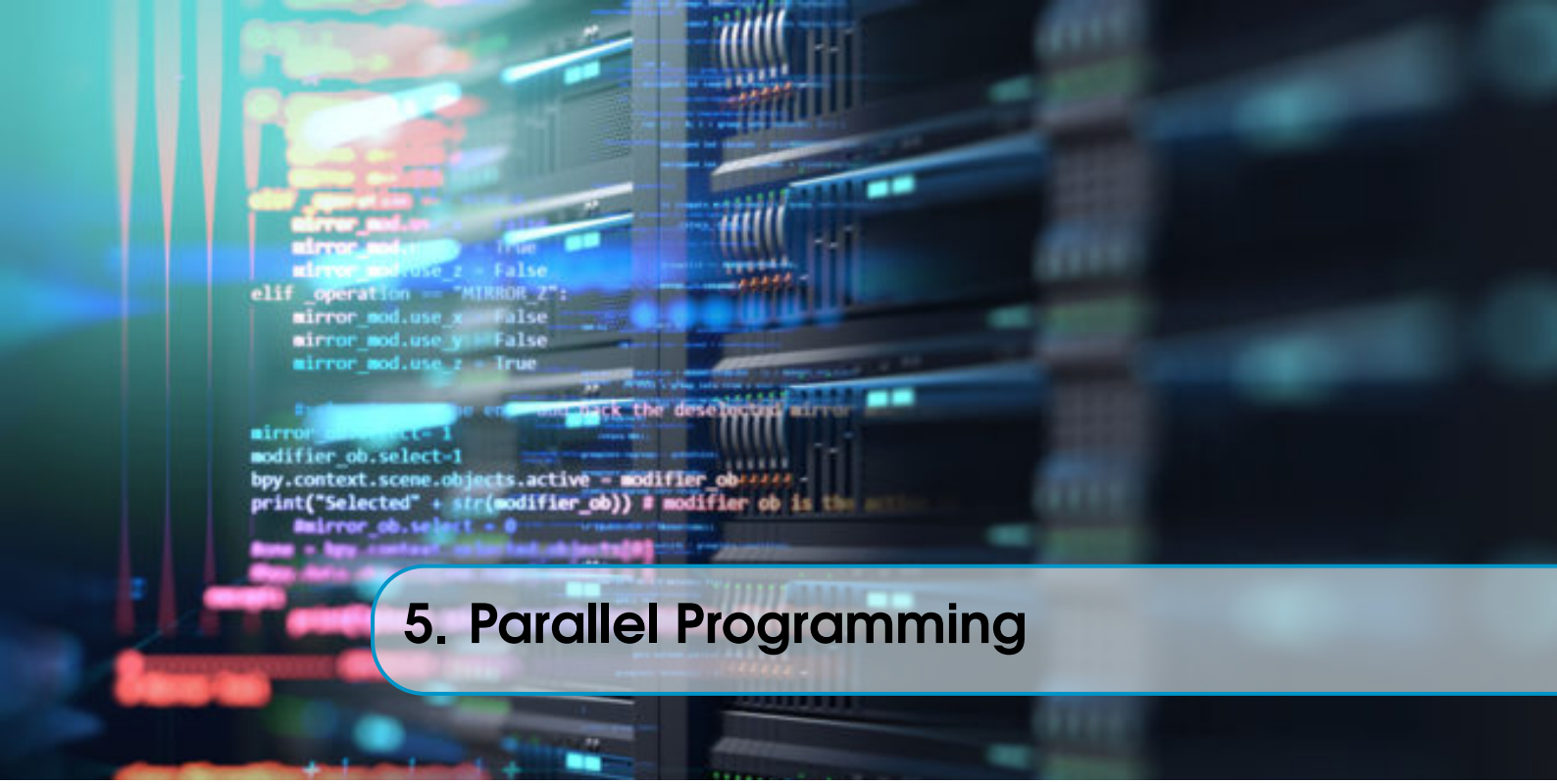
The SaaS concept has several connections to earlier technologies. For instance, after the mainframe and workstation eras, the so-called *thin client* idea was briefly popular. Here, the user would have a workstation rather than a terminal, yet work on data stored on a central server. One product along these lines was Sun's *Sun Ray* (circa 1999) where users relied on a smartcard to establish their local environment on an arbitrary, otherwise stateless, workstation.

4.8.2 Heterogeneous computing

You have now seen several computing models: single core, shared memory multicore, distributed memory clusters, GPUs. These models all have in common that, if there is more than one instruction stream active, all streams are interchangeable. With regard to GPUs we need to refine this statement: all instruction stream *on the GPU* are interchangeable. However, a GPU is not a standalone device, but can be considered a *co-processor* to a *host processor*.

If we want to let the host perform useful work while the co-processor is active, we now have two different instruction streams or types of streams. This situation is known as *heterogeneous computing*. In the GPU case, these instruction streams are even programmed by a slightly different mechanisms – using *CUDA* for the GPU – but this need not be the case: the Intel Multi

Integrated Cores (MIC) architecture is programmed in ordinary C.



5. Parallel Programming

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

There are various approaches to parallel programming. First of all, there does not seem to be any hope of a *parallelizing compiler* that can automagically transform a sequential program into a parallel one. Apart from the problem of figuring out which operations are independent, the main problem is that the problem of locating data in a parallel context is very hard. A compiler would need to consider the whole code, rather than a subroutine at a time. Even then, results have been disappointing.

More productive is the approach where the user writes mostly a sequential program, but gives some indications about what computations can be parallelized, and how data should be distributed. Indicating parallelism of operations explicitly is done in OpenMP (section 5.2); indicating the data distribution and leaving parallelism to the compiler and runtime is the basis for PGAS languages. Such approaches work best with shared memory.

By far the hardest way to program in parallel, but with the best results in practice, is to expose the parallelism to the programmer and let the programmer manage everything explicitly. This approach is necessary in the case of distributed memory programming. We will have a general discussion of distributed programming in section 5.3.1; section 5.3.3 will discuss the MPI library.

5.1 Thread parallelism

As a preliminary to OpenMP (section 5.2), we will briefly go into 'threads'.

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

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

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

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

5.1.1 The fork-join mechanism

Threads are dynamic, in the sense that they can be created during program execution. (This is different from the MPI model, where every processor run one process, and they are all created and destroyed at the same time.) When a program starts, there is one thread active: the *master thread*. Other threads are created by *thread spawning*, and the master thread can wait for their completion. This is known as the *fork-join* model; it is illustrated in figure 5.1. A group

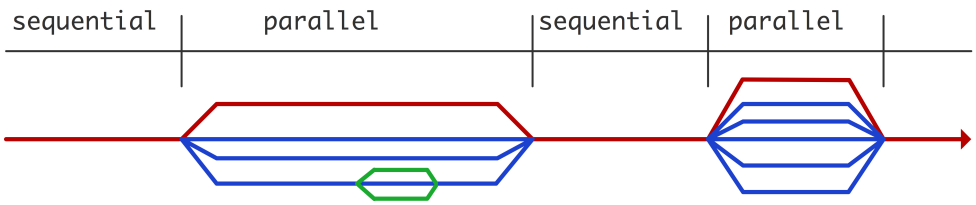


Figure 5.1 Thread creation and deletion during parallel execution

of threads that is forked from the same thread and active simultaneously is known as a *thread team*.

5.1.2 Hardware support for threads

Threads as they were described above are a software construct. Threading was possible before parallel computers existed; they were for instance used to handle independent activitives in an OS. In the absence of parallel hardware, the OS would handle the threads through *multitasking* or *time slicing*: each thread would regularly get to use the CPU for a fraction of a second. (Technically, the Linux kernel treads processes and threads though the *task* concept; tasks are kept in a list, and are regularly activated or de-activated.)

This can lead to higher processor utilization, since the instructions of one thread can be processed while another thread is waiting for data. (On traditional

CPUs, switching between threads is somewhat expensive (an exception is the *hyperthreading* mechanism) but on GPU it is not, and in fact they *need* many threads to attain high performance.)

On modern *multicore* processors there is an obvious way of supporting threads: having one thread per core gives a parallel execution that uses your hardware efficiently. The shared memory allows the threads to all see the same data. This can also lead to problems; see section 5.1.5.

5.1.3 Threads example

The following example[9] is a clear illustration of the *fork-join* model. It uses the *pthreads* library to spawn a number of tasks that all update a global counter. Since threads share the same memory space, they indeed see and update the same memory location.

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

int sum=0;

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

#define NTHREADS 50
int main() {
  int i;
  pthread_t threads[NTHREADS];
  printf("forking\n");
  for (i=0; i<NTHREADS; i++)
    if (pthread_create(threads+i,NULL,&adder,NULL)!=0) return i+1;
  printf("joining\n");
  for (i=0; i<NTHREADS; i++)
    if (pthread_join(threads[i],NULL)!=0) return NTHREADS+i+1;
  printf("Sum computed: %d\n",sum);

  return 0;
}
```

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

[9] This is strictly Unix-centric and will not work on Windows.

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

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

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

```
pthread_mutex_t lock;

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

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

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

For more information on pthreads, see for instance `https://computing.llnl.gov/tutorials/pthreads`.

5.1.4 Contexts

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

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

The totality of all data that a thread can access is called its *context*. It contains private and shared data, as well as temporary results of computations that the thread is working on[10].

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

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

[10] It also contains the program counter and stack pointer. If you don't know what those are, don't worry.

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

5.1.5 Race conditions, thread safety, and atomic operations

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

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

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

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

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

Such a scenario, where the final result depends on which thread executes first, is known as a *race condition* or *data race*.

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

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

Here the products are truly independent, so we could choose to have the loop iterations do them in parallel, for instance by their own threads. However, all threads need to update the same variable `sum`.

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

There are essentially two ways of solving this problem. One is that we declare such updates of a shared variable a *critical section* of code. This means that the instructions in the critical section (in the inner product example 'read `sum` from memory, update it, write back to memory') can be executed by only

one thread at a time. In particular, they need to be executed entirely by one thread before any other thread can start them so the ambiguity problem above will not arise. Of course, the above code fragment is so common that systems like OpenMP (section 5.2) have a dedicated mechanism for it, by declaring it a *reduction* operation.

Critical sections can for instance be implemented through the *semaphore* mechanism [14]. Surrounding each critical section there will be two atomic operations controlling a semaphore, a sign post. The first process to encounter the semaphore will lower it, and start executing the critical section. Other processes see the lowered semaphore, and wait. When the first process finishes the critical section, it executes the second instruction which raises the semaphore, allowing one of the waiting processes to enter the critical section.

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

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

One implementation of this is *transactional memory*, where the hardware itself supports atomic operations; the term derives from database transactions, which have a similar integrity problem. In transactional memory, a process will perform a normal memory update, unless the processor detects a conflict with an update from another process. In that case, the updates ('transactions') are aborted and retried with one processor locking the memory and the other waiting for the lock. This is an elegant solution; however, aborting the transaction may carry a certain cost of *pipeline flushing* (section 3.2.5) and cache line invalidation.

5.1.6 Memory models and sequential consistency

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

As an example, consider

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

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

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

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

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

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

In a *relaxed memory model* it is possible to get a result that is not sequentially consistent. Suppose, in the above example, the instructions A ← 1, B ← 1 are executed first, but these changes are not immediately visible to the other threads. Then subsequent instructions x ← B, y ← A can receive the result $x = 0, y = 0$, which was not possible under the sequentially consistent model above.

Sequential consistency implies that

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

should have the same effect as

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

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

Exercise 5.1 In section 5.1.5 you saw an example that needed a critical section to get the right final result. Argue that having a critical section is not enough for sequential consistency.

- Write a piece of sequential code that, when executed in parallel, corre-

sponds to the example in section 5.1.5.
- Show that, using a critical section, there are two execution orderings that give the correct result.
- Show that one of these orderings is not sequentially consistent.

5.1.7 Affinity

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

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

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

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

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

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

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

thread	socket 0	socket 1
0	0	
1	1	

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

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

thread	socket 0	socket 1
0	0	
1		4

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

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

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

versus

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

5.1.8 Hyperthreading versus multi-threading

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

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

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

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

5.2 OpenMP

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

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

OpenMP is not a language, but an extension to the existing C and Fortran languages. It mostly operates by inserting directives into source code, which are interpreted by the compiler. It also has a modest number of library calls, but these are not the main point, unlike in MPI (section 5.3.3). Finally, there is a runtime system that manages the parallel execution.

OpenMP has an important advantage over MPI in its programmability:

it is possible to start with a sequential code and transform it by *incremental parallelization*. By contrast, turning a sequential code into a distributed memory MPI program is an all-or-nothing affair.

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

For more information about OpenMP, see [10] and `http://openmp.org/wp/`.

5.2.1 OpenMP examples

The simplest example of OpenMP use is the parallel loop.

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

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

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

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

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

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

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

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

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

As you see, a sequential code can be easily parallelized this way.

The assignment of iterations to threads is done by the runtime system, but the user can guide this assignment. We are mostly concerned with the case where there are more iterations than threads: if there are P threads and N iterations and $N > P$, how is iteration i going to be assigned to a thread?

The simplest assignment uses *round-robin task scheduling*, a *static scheduling* strategy where thread p gets iterations $p \times (N/P), \ldots, (p+1) \times (N/P) - 1$. This has the advantage that if some data is reused between iterations, it will stay in the data cache of the processor executing that thread. On the other hand, if the iterations differ in the amount of work involved, the process may suffer from *load unbalance* with static scheduling. In that case, a *dynamic scheduling* strategy would work better, where each thread starts work on the next unprocessed iteration as soon as it finishes its current iteration.

You can control OpenMP scheduling of loop iterations with the `schedule` keyword; its values include `static` and `dynamic`. It is also possible to indicate a `chunksize`, which controls the size of the block of iterations that gets assigned together to a thread. If you omit the chunksize, OpenMP will divide the iterations into as many blocks as there are threads.

Exercise 5.2 Let's say there are t threads, and your code looks like

```
for (i=0; i<N; i++) {
  a[i] = // some calculation
}
```

If you specify a chunksize of 1, iterations $0, t, 2t, \ldots$ go to the first thread, $1, 1+t, 1+2t, \ldots$ to the second, et cetera. Discuss why this is a bad strategy from a performance point of view. Hint: look up the definition of *false sharing*. What would be a good chunksize?

5.3 Distributed memory programming through message passing

While OpenMP programs, and programs written using other shared memory paradigms, still look very much like sequential programs, this does not hold true for message passing code. Before we discuss the Message Passing Interface (MPI) library in some detail, we will take a look at this shift the way parallel code is written.

5.3.1 The global versus the local view in distributed programming

There can be a marked difference between how a parallel algorithm looks to an observer, and how it is actually programmed. Consider the case where we have an array of processors $\{P_i\}_{i=0..p-1}$, each containing one element of the arrays x and y, and P_i computes

$$\begin{cases} y_i \leftarrow y_i + x_{i-1} & i > 0 \\ y_i \text{ unchanged} & i = 0 \end{cases} \tag{5.1}$$

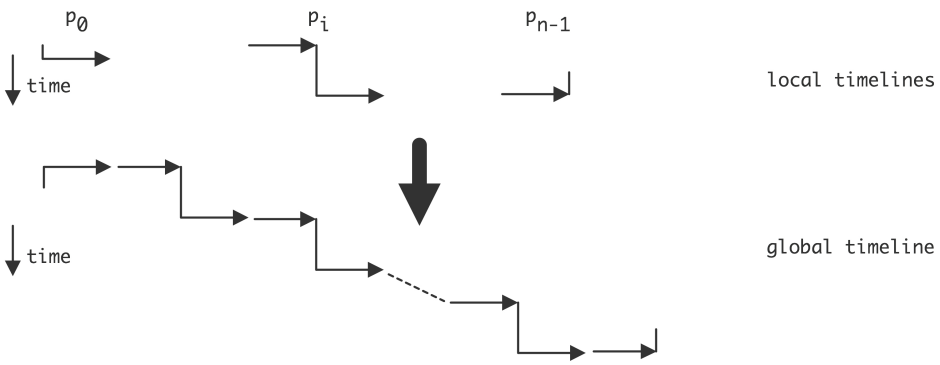


Figure 5.2 Local and resulting global view of an algorithm for sending data to the right

The global description of this could be

- Every processor P_i except the last sends its x element to P_{i+1};
- every processor P_i except the first receive an x element from their neighbour P_{i-1}, and
- they add it to their y element.

However, in general we can not code in these global terms. In the SPMD model (section 4.3.2) each processor executes the same code, and the overall algorithm is the result of these individual behaviours. The local program has access only to local data – everything else needs to be communicated with send and receive operations – and the processor knows its own number.

One possible way of writing this would be

- If I am processor 0 do nothing. Otherwise receive a y element from the left, add it to my x element.
- If I am the last processor do nothing. Otherwise send my y element to the right.

At first we look at the case where sends and receives are so-called *blocking communication* instructions: a send instruction does not finish until the sent item is actually received, and a receive instruction waits for the corresponding send. This means that sends and receives between processors have to be carefully paired. We will now see that this can lead to various problems on the way to an efficient code.

The above solution is illustrated in figure 5.2, where we show the local timelines depicting the local processor code, and the resulting global behaviour. You see that the processors are not working at the same time: we get *serialized execution*.

What if we reverse the send and receive operations?

- If I am not the last processor, send my x element to the right;
- If I am not the first processor, receive an x element from the left and add it to my y element.

This is illustrated in figure 5.3 and you see that again we get a serialized execution, except that now the processors are activated right to left.

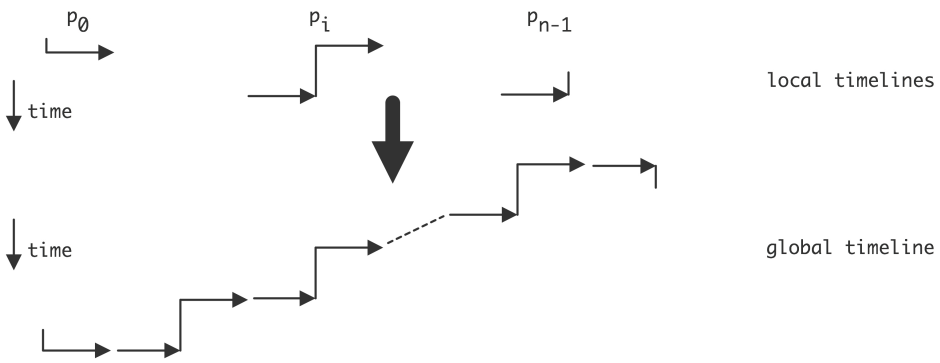


Figure 5.3 Local and resulting global view of an algorithm for sending data to the right

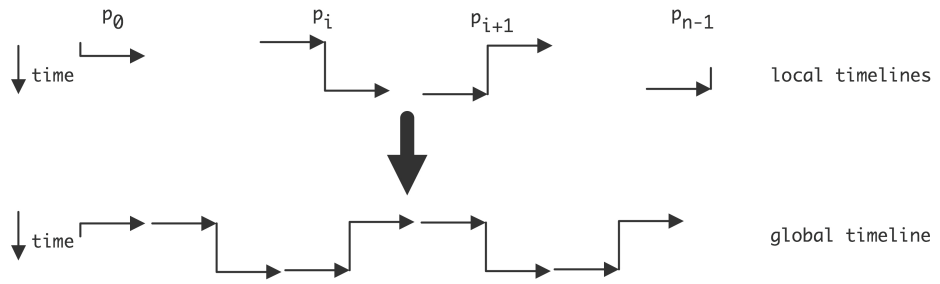


Figure 5.4 Local and resulting global view of an algorithm for sending data to the right

If the algorithm in equation 5.1 had been cyclic:

$$\begin{cases} y_i \leftarrow y_i + x_{i-1} & i = 1 \ldots n-1 \\ y_0 \leftarrow y_0 + x_{n-1} & i = 0 \end{cases} \tag{5.2}$$

the problem would be even worse. Now the last processor can not start its receive since it is blocked sending x_{n-1} to processor 0. This situation, where the program can not progress because every processor is waiting for another, is called *deadlock*.

The solution to getting an efficient code is to make as much of the communication happen simultaneously as possible. After all, there are no serial dependencies in the algorithm. Thus we program the algorithm as follows:

- If I am an odd numbered processor, I send first, then receive;
- If I am an even numbered processor, I receive first, then send.

This is illustrated in figure 5.4, and we see that the execution is now parallel.

Exercise 5.3 Take another look at figure 4.3 of a parallel reduction. The basic actions are:

- receive data from a neighbour
- add it to your own data
- send the result on.

As you see in the diagram, there is at least one processor who does not send data on, and others may do a variable number of receives before they send their result on.

Write node code so that an SPMD program realizes the distributed

reduction. Hint: write each processor number in binary. The algorithm uses a number of steps that is equal to the length of this bitstring.

- Assuming that a processor receives a message, express the distance to the origin of that message in the step number.
- Every processor sends at most one message. Express the step where this happens in terms of the binary processor number.

5.3.2 Blocking and non-blocking communication

The reason for blocking instructions is to prevent accumulation of data in the network. If a send instruction were to complete before the corresponding receive started, the network would have to store the data somewhere in the mean time. Consider a simple example:

```
buffer = ... ;  // generate some data
send(buffer,0); // send to processor 0
buffer = ... ;  // generate more data
send(buffer,1); // send to processor 1
```

After the first send, we start overwriting the buffer. If the data in it hasn't been received, the first set of values would have to be buffered somewhere in the network, which is not realistic. By having the send operation block, the data stays in the sender's buffer until it is guaranteed to have been copied to the recipient's buffer.

One way out of the problem of sequentialization or deadlock that arises from blocking instruction is the use of *non-blocking communication* instructions, which include explicit buffers for the data. With non-blocking send instruction, the user needs to allocate a buffer for each send, and check when it is safe to overwrite the buffer.

```
buffer0 = ... ;   // data for processor 0
send(buffer0,0);  // send to processor 0
buffer1 = ... ;   // data for processor 1
send(buffer1,1);  // send to processor 1
...
// wait for completion of all send operations.
```

5.3.3 The MPI library

If OpenMP is the way to program shared memory, Message Passing Interface (MPI) [36] is the standard solution for programming distributed memory. MPI ('Message Passing Interface') is a specification for a library interface for moving data between processes that do not otherwise share data. The MPI routines can be divided roughly in the following categories:

- Process management. This includes querying the parallel environment and constructing subsets of processors.
- Point-to-point communication. This is a set of calls where two processes interact. These are mostly variants of the send and receive calls.

- Collective calls. In these routines, all processors (or the whole of a specified subset) are involved. Examples are the *broadcast* call, where one processor shares its data with every other processor, or the *gather* call, where one processor collects data from all participating processors.

Let us consider how the OpenMP examples can be coded in MPI[11]. First of all, we no longer allocate

```
double a[ProblemSize];
```

but

```
double a[LocalProblemSize];
```

where the local size is roughly a $1/P$ fraction of the global size. (Practical considerations dictate whether you want this distribution to be as evenly as possible, or rather biased in some way.)

The parallel loop is trivially parallel, with the only difference that it now operates on a fraction of the arrays:

```
for (i=0; i<LocalProblemSize; i++) {
  a[i] = b[i];
}
```

However, if the loop involves a calculation based on the iteration number, we need to map that to the global value:

```
for (i=0; i<LocalProblemSize; i++) {
  a[i] = b[i]+f(i+MyFirstVariable);
}
```

(We will assume that each process has somehow calculated the values of `LocalProblemSize` and `MyFirstVariable`.) Local variables are now automatically local, because each process has its own instance:

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

However, shared variables are harder to implement. Since each process has its own data, the local accumulation has to be explicitly assembled:

```
for (i=0; i<LocalProblemSize; i++) {
  s = s + a[i]*b[i];
}
MPI_Allreduce(s,globals,1,MPI_DOUBLE,MPI_SUM);
```

[11] This is not a course in MPI programming, and consequently the examples will leave out many details of the MPI calls. If you want to learn MPI programming, consult for instance [37], [22], [20].

The 'reduce' operation sums together all local values `s` into a variable `globals` that receives an identical value on each processor. This is known as a *collective operation*.

Let us make the example slightly more complicated:

```
for (i=0; i<ProblemSize; i++) {
  if (i==0)
    a[i] = (b[i]+b[i+1])/2
  else if (i==ProblemSize-1)
    a[i] = (b[i]+b[i-1])/2
  else
    a[i] = (b[i]+b[i-1]+b[i+1])/3
}
```

If we had shared memory, we could write the following parallel code:

```
for (i=0; i<LocalProblemSize; i++) {
  bleft = b[i-1]; bright = b[i+1];
  a[i] = (b[i]+bleft+bright)/3
}
```

To turn this into valid distributed memory code, first we account for the fact that `bleft` and `bright` need to be obtained from a different processor for `i==0` (`bleft`), and for `i==LocalProblemSize-1` (`bright`). We do this with a exchange operation with our left and right neighbour processor:

```
// get bfromleft and bfromright from neighbour processors, then
for (i=0; i<LocalProblemSize; i++) {
  if (i==0) bleft=bfromleft;
    else bleft = b[i-1]
  if (i==LocalProblemSize-1) bright=bfromright;
    else bright = b[i+1];
  a[i] = (b[i]+bleft+bright)/3
}
```

Obtaining the neighbour values is done as follows. First we need to ask our processor number, so that we can start a communication with the processor with a number one higher and lower.

```
MPI_Comm_rank(MPI_COMM_WORLD,&myTaskID);
MPI_Sendrecv
  (/* to be sent:  */ &b[LocalProblemSize-1],
    /* destination  */ myTaskID+1,
    /* to be recvd: */ &bfromleft,
    /* source:      */ myTaskID-1,
    /* some parameters omitted */
  );
MPI_Sendrecv(&b[0],myTaskID-1,
    &bfromright, /* ... */ );
```

There are still two problems with this code. First, the sendrecv operations need exceptions for the first and last processors. This can be done elegantly as follows:

```
MPI_Comm_rank(MPI_COMM_WORLD,&myTaskID);
MPI_Comm_size(MPI_COMM_WORLD,&nTasks);
if (myTaskID==0) leftproc = MPI_PROC_NULL;
  else leftproc = myTaskID-1;
if (myTaskID==nTasks-1) rightproc = MPI_PROC_NULL;
  else rightproc = myTaskID+1;
MPI_Sendrecv( &b[LocalProblemSize-1], &bfromleft,  rightproc );
MPI_Sendrecv( &b[0],                  &bfromright, leftproc);
```

MPI gets complicated if different processes need to take different actions, for example, if one needs to send data to another. The problem here is that each process executes the same executable, so it needs to contain both the send and the receive instruction, to be executed depending on what the rank of the process is.

```
if (myTaskID==0) {
  MPI_Send(myInfo,1,MPI_INT,/* to: */ 1,/* labeled: */,0,
    MPI_COMM_WORLD);
} else {
  MPI_Recv(myInfo,1,MPI_INT,/* from: */ 0,/* labeled: */,0,
    /* not explained here: */&status,MPI_COMM_WORLD);
}
```

5.3.4 Blocking

Although MPI is sometimes called the 'assembly language of parallel programming', for its perceived difficulty and level of explicitness, it is not all that hard to learn, as evinced by the large number of scientific codes that use it. The main issues that make MPI somewhat intricate to use are buffer management and blocking semantics.

These issues are related, and stem from the fact that, ideally, data should not be in two places at the same time. Let us briefly consider what happens if processor 1 sends data to processor 2. The safest strategy is for processor 1 to execute the send instruction, and then wait until processor 2 acknowledges that the data was successfully received. This means that processor 1 is temporarily blocked until processor 2 actually executes its receive instruction, and the data has made its way through the network. This is the standard behaviour of the `MPI_Send` and `MPI_Recv` calls, which are said to use *blocking communication*.

Alternatively, processor 1 could put its data in a buffer, tell the system to make sure that it gets sent at some point, and later checks to see that the buffer is safe to reuse. This second strategy is called *non-blocking communication*, and it requires the use of a temporary buffer.

5.3.5 Collective operations

In the above examples, you saw the `MPI_Allreduce` call, which computed a global sum and left the result on each processor. There is also a local version `MPI_Reduce` which computes the result only on one processor. These calls are examples of *collective operations* or collectives. The collectives are:

reduction : each processor has a data item, and these items need to be combined arithmetically with an addition, multiplication, max, or min operation. The result can be left on one processor, or on all, in which case we call this an **allreduce** operation.

broadcast : one processor has a data item that all processors need to receive.

gather : each processor has a data item, and these items need to be collected in an array, without combining them in an operations such as an addition. The result can be left on one processor, or on all, in which case we call this an **allgather**.

scatter : one processor has an array of data items, and each processor receives one element of that array.

all-to-all : each processor has an array of items, to be scattered to all other processors.

Collective operations are blocking (see section 5.3.4), although MPI 3.0 (which is currently only a draft) will have non-blocking collectives.

5.3.6 Non-blocking communication

In a simple computer program, each instruction takes some time to execute, in a way that depends on what goes on in the processor. In parallel programs the situation is more complicated. A send operation, in its simplest form, declares that a certain buffer of data needs to be sent, and program execution will then stop until that buffer has been safely sent and received by another processor. This sort of operation is called a *non-local operation* since it depends on the actions of other processes, and a *blocking communication* operation since execution will halt until a certain event takes place.

Blocking operations have the disadvantage that they can lead to *deadlock*. In the context of message passing this describes the situation that a process is waiting for an event that never happens; for instance, it can be waiting to receive a message and the sender of that message is waiting for something else. You can easily recognize that deadlock occurs if two processes are waiting for each other, or more generally, if you have a cycle of processes where each is waiting for the next process in the cycle. Example:

```
if ( /* this is process 0 */ )
  // wait for message from 1
else if ( /* this is process 1 */ )
  // wait for message from 0
```

A block receive here leads to deadlock. Even without deadlock, they can lead to considerable *idle time* in the processors, as they wait without performing any useful work. On the other hand, they have the advantage that it is clear when

the buffer can be reused: after the operation completes, there is a guarantee that the data has been safely received at the other end.

The blocking behaviour can be avoided, at the cost of complicating the buffer semantics, by using *non-blocking communication* operations. A non-blocking send (`MPI_Isend`) declares that a data buffer needs to be sent, but then does not wait for the completion of the corresponding receive. There is a second operation `MPI_Wait` that will actually block until the receive has been completed. The advantage of this decoupling of sending and blocking is that it now becomes possible to write:

```
MPI_ISend(somebuffer,&handle); // start sending, and
    // get a handle to this particular communication
{ ... }  // do useful work on local data
MPI_Wait(handle); // block until the communication is completed;
{ ... }  // do useful work on incoming data
```

With a little luck, the local operations take more time than the communication, and you have completely eliminated the communication time.

In addition to non-blocking sends, there are non-blocking receives. A typical piece of code then looks like

```
MPI_ISend(sendbuffer,&sendhandle);
MPI_IReceive(recvbuffer,&recvhandle);
{ ... }  // do useful work on local data
MPI_Wait(sendhandle); Wait(recvhandle);
{ ... }  // do useful work on incoming data
```

5.3.7 MPI version 1 and 2 and 3

The first MPI standard [35] had a number of notable omissions, which are included in the MPI 2 standard [21]. One of these concerned parallel input/output: there was no facility for multiple processes to access the same file, even if the underlying hardware would allow that. A separate project MPI-I/O has now been rolled into the MPI-2 standard. We will discuss parallel I/O in this book.

A second facility missing in MPI, though it was present in *PVM* [29], [1] which predates MPI, is process management: there is no way to create new processes and have them be part of the parallel run.

Finally, MPI-2 has support for one-sided communication: one process puts data into the memory of another, without the receiving process doing an actual receive instruction. We will have a short discussion in section 5.3.8 below.

With MPI-3 the standard has gained a number of new features, such as non-blocking collectives, neighbourhood collectives, and a profiling interface. The one-sided mechanisms have also been updated.

5.3.8 One-sided communication

The MPI way of writing matching send and receive instructions is not ideal for a number of reasons. First of all, it requires the programmer to give the same

data description twice, once in the send and once in the receive call. Secondly, it requires a rather precise orchestration of communication if deadlock is to be avoided; the alternative of using asynchronous calls is tedious to program, requiring the program to manage a lot of buffers. Lastly, it requires a receiving processor to know how many incoming messages to expect, which can be tricky in irregular applications. Life would be so much easier if it was possible to pull data from another processor, or conversely to put it on another processor, without that other processor being explicitly involved.

This style of programming is further encouraged by the existence of *Remote Direct Memory Access (RDMA)* support on some hardware. An early example was the *Cray T3E*. These days, one-sided communication is widely available through its incorporation in the MPI-2 library; section 5.3.7.

Let us take a brief look at one-sided communication in MPI-2, using averaging of array values as an example:

$$\forall_i\colon a_i \leftarrow (a_i + a_{i-1} + a_{i+1})/3.$$

The MPI parallel code will look like

```
// do data transfer
a_local = (a_local+left+right)/3
```

It is clear what the transfer has to accomplish: the `a_local` variable needs to become the `left` variable on the processor with the next higher rank, and the `right` variable on the one with the next lower rank.

First of all, processors need to declare explicitly what memory area is available for one-sided transfer, the so-called 'window'. In this example, that consists of the `a_local`, `left`, and `right` variables on the processors:

```
MPI_Win_create(&a_local,...,&data_window);
MPI_Win_create(&left,....,&left_window);
MPI_Win_create(&right,....,&right_window);
```

The code now has two options: it is possible to push data out

```
target = my_tid-1;
MPI_Put(&a_local,...,target,right_window);
target = my_tid+1;
MPI_Put(&a_local,...,target,left_window);
```

or to pull it in

```
data_window = a_local;
source = my_tid-1;
MPI_Get(&right,...,data_window);
source = my_tid+1;
MPI_Get(&left,...,data_window);
```

The above code will have the right semantics if the Put and Get calls are blocking; see section 5.3.4. However, part of the attraction of one-sided communication is that it makes it easier to express communication, and for this, a non-blocking semantics is assumed.

The problem with non-blocking one-sided calls is that it becomes necessary to ensure explicitly that communication is successfully completed. For instance, if one processor does a one-sided *put* operation on another, the other processor has no way of checking that the data has arrived, or indeed that transfer has begun at all. Therefore it is necessary to insert a global barrier in the program, for which every package has its own implementation. In MPI-2 the relevant call is the `MPI_Win_fence` routine. These barriers in effect divide the program execution in *supersteps*.

5.4 Data dependencies

If two statements refer to the same data item, we say that there is a *data dependency* between the statements. Such dependencies limit the extent to which the execution of the statements can be rearranged. The study of this topic probably started in the 1960s, when processors could execute statements *out of order* to increase throughput. The re-ordering of statements was limited by the fact that the execution had to obey the *program order* semantics: the result had to be as if the statements were executed strictly in the order in which they appear in the program.

These issues of statement ordering, and therefore of data dependencies, arise in several ways:

- A *parallelizing compiler* has to analyze the source to determine what transformations are allowed;
- if you parallelize a sequential code with OpenMP directives, you have to perform such an analysis yourself.

Here are two types of activity that require such an analysis:

- When a loop is parallelized, the iterations are no longer executed in their program order, so we have to check for dependencies.
- The introduction of tasks also means that parts of a program can be executed in a different order from in which they appear in a sequential execution.

The easiest case of dependency analysis is that of detecting that loop iterations can be executed independently. Iterations are of course independent if a data item is read in two different iterations, but if the same item is read in one iteration and written in another, or written in two different iterations, we need to do further analysis.

Analysis of *data dependencies* can be performed by a compiler, but compilers take, of necessity, a conservative approach. This means that iterations may be independent, but can not be recognized as such by a compiler. Therefore, OpenMP shifts this responsibility to the programmer.

We will now discuss data depencies in some detail.

5.4.1 Types of data dependencies

The three types of dependencies are:

- flow dependencies, or 'read-after-write';
- anti dependencies, or 'write-after-read'; and
- output dependencies, or 'write-after-write'.

These dependencies can be studied in scalar code, and in fact compilers do this to determine whether statements can be rearranged, but we will mostly be concerned with their appearance in loops, since in scientific computation much of the work appears there.

Flow dependencies

Flow dependencies, or read-afer-write, are not a problem if the read and write occur in the same loop iteration:

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

On the other hand, if the read happens in a later iteration, there is no simple way to parallelize the loop:

```
for (i=0; i<N; i++) {
  .... = ... x[i] ... ;
  x[i+1] = .... ;
}
```

This usually requires rewriting the code.

Anti dependencies

The simplest case of an *anti dependency* or write-after-read is a reduction:

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

This can be dealt with by explicit declaring the loop to be a reduction.

If the read and write are on an array the situation is more complicated. The iterations in this fragment

```
for (i=0; i<N; i++) {
  x[i] = ... x[i+1] ... ;
}
```

can not be executed in arbitrary order as such. However, conceptually there is no dependency. We can solve this by introducing a temporary array:

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

```
for (i=0; i<N; i++) {
  x[i] = ... xtmp[i+1] ... ;
}
```

This is an example of a transformation that a compiler is unlikely to perform, since it can greatly affect the memory demands of the program. Thus, this is left to the programmer.

Output dependencies

The case of an *output dependency* or write-after-write does not occur by itself: if a variable is written twice in sequence without an intervening read, the first write can be removed without changing the meaning of the program. Thus, this case reduces to a flow dependency.

Other output dependencies can easily be removed. In the following code, `t` can be declared private, thereby removing the dependency.

```
for (i=0; i<N; i++) {
  t = f(i)
  s += t*t;
}
```

If the final value of `t` is wanted, the lastprivate can be used in OpenMP.

PART III
OPEN MULTI-PROCESSING

6. OpenMP

The following chapter contains reference material for using the OpenMP directives in parallel programming. This chapter is an extension of section 5.2, and intended as reference only.

6.1 The OpenMP Model

This section explains the basic concepts of OpenMP, and helps you get started on running your first OpenMP program. We start by establishing a mental picture of the hardware and software that OpenMP targets.

6.1.1 Target hardware

Modern computers have a multi-layered design. Maybe you have access to a cluster, and maybe you have learned how to use MPI to communicate between cluster nodes. OpenMP, the topic of this chapter, is concerned with a single *cluster node* or *motherboard*, and getting the most out of the available parallelism available there.

In Figure 3.9 on page 56 we saw a typical design of a node; within one enclosure you find two *sockets*: single processor chips. Your personal laptop or computer will probably have one socket, most supercomputers have nodes with two or four sockets.

To see where OpenMP operates we need to dig into the sockets. Figure 3.1 on page 40 showed a picture of an *Intel Sandybridge* socket. You recognize a structure with eight *cores*: independent processing units, that all have access to the same memory.

To summarize the structure of the architecture that OpenMP targets:

- A node has up to four sockets;
- each socket has up to 60 cores;

- each core is an independent processing unit, with access to all the memory on the node.

6.1.2 Target software

OpenMP is based on on two concepts: the use of *threads* and the *fork/join model* of parallelism. For now you can think of a thread as a sort of process: the computer executes a sequence of instructions. The fork/join model says that a thread can split itself ('fork') into a number of threads that are identical copies. At some point these copies go away and the original thread is left ('join'), but while the *team of* threads created by the fork exists, you have parallelism available to you. The part of the execution between fork and join is known as a *parallel region*.

In section 5.1 earlier, we already saw an example of this: a thread forks into a team of threads, and these threads themselves can fork again.

The threads that are forked are all copies of the *master thread*: they have access to all that was computed so far; this is their *shared data*. Of course, if the threads were completely identical the parallelism would be pointless, so they also have private data, and they can identify themselves: they know their thread number. This allows you to do meaningful parallel computations with threads.

This brings us to the third important concept: that of *work sharing* constructs. In a team of threads, initially there will be replicated execution; a work sharing construct divides available parallelism over the threads.

> So there you have it: OpenMP uses teams of threads, and inside a parallel region the work is distributed over the threads with a work sharing construct. Threads can access shared data, and they have some private data.

An important difference between OpenMP and MPI is that parallelism in OpenMP is dynamically activated by a thread spawning a team of threads. Furthermore, the number of threads used can differ between parallel regions, and threads can create threads recursively. This is known as as *dynamic mode*. By contrast, in an MPI program the number of running processes is (mostly) constant throughout the run, and determined by factors external to the program.

6.1.3 About threads and cores

OpenMP programming is typically done to take advantage of *multicore* processors. Thus, to get a good speedup you would typically let your number of threads be equal to the number of cores. However, there is nothing to prevent you from creating more threads: the operating system will use *time slicing* to let them all be executed. You just don't get a speedup beyond the number of actually available cores.

On some modern processors there are *hardware threads*, meaning that a core can actually let more than thread be executed, with some speedup over the

single thread. To use such a processor efficiently you would let the number of OpenMP threads be 2× or 4× the number of cores, depending on the hardware.

6.1.4 About thread data

In most programming languages, visibility of data is governed by rules on the *scope of variables*: a variable is declared in a block, and it is then visible to any statement in that block and blocks with a *lexical scope* contained in it, but not in surrounding blocks:

```
main () {
  // no variable 'x' define here
  {
    int x = 5;
    if (somecondition) { x = 6; }
    printf("x=%e\n",x); // prints 5 or 6
  }
  printf("x=%e\n",x); // syntax error: 'x' undefined
}
```

In C, you can redeclare a variable inside a nested scope:

```
{
  int x;
  if (something) {
    double x; // same name, different entity
  }
  x = ... // this refers to the integer again
}
```

Doing so makes the outer variable inaccessible.

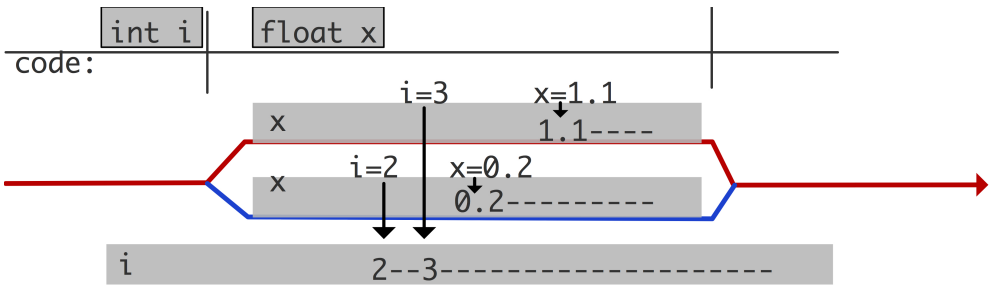


Figure 6.1 Locality of variables in threads

In OpenMP the situation is a bit more tricky because of the threads. When a team of threads is created they can all see the data of the master thread. However, they can also create data of their own. This is illustrated in figure 6.1. We will go into the details later.

6.2 Compiling and running an OpenMP program

6.2.1 Compiling

Your file or needs to contain

```
#include "omp.h"
```

in C.

OpenMP is handled by extensions to your regular compiler, typically by adding an option to your commandline:

```
# gcc
gcc -o foo foo.c -fopenmp
# Intel compiler
icc -o foo foo.c -openmp
```

If you have separate compile and link stages, you need that option in both.

6.2.2 Running an OpenMP program

You run an OpenMP program by invoking it the regular way (for instance `./a.out`), but its behaviour is influenced by some *OpenMP environment variables*. The most important one is `OMP_NUM_THREADS`:

```
export OMP_NUM_THREADS=8
```

which sets the number of threads that a program will use. See section C.1 for a list of all environment variables.

6.3 Your first OpenMP program

In this section you will see just enough of OpenMP to write a first program and to explore its behaviour. For this we need to introduce a couple of OpenMP language constructs. They will all be discussed in much greater detail in later chapters.

6.3.1 Directives

OpenMP is not magic, so you have to tell it when something can be done in parallel. This is mostly done through *directives*; additional specifications can be done through library calls.

In C/C++ the *pragma* mechanism is used: annotations for the benefit of the compiler that are otherwise not part of the language. This looks like:

```
#pragma omp somedirective clause(value,othervalue)
  parallel statement;

#pragma omp somedirective clause(value,othervalue)
 {
  parallel statement 1;
  parallel statement 2;
 }
```

with

- the `#pragma omp` *sentinel* to indicate that an OpenMP directive is coming;
- a directive, such as `parallel`;

- and possibly clauses with values.
- After the directive comes either a single statement or a block in *curly braces*.

Directives in C/C++ are case-sensitive. Directives can be broken over multiple lines by escaping the line end.

6.3.2 Parallel regions

The simplest way to create parallelism in OpenMP is to use the *parallel* pragma. A block preceded by the `omp parallel` pragma is called a *parallel region*; it is executed by a newly created team of threads. This is an instance of the *SPMD* model: all threads execute the same segment of code.

```
#pragma omp parallel
{
  // this is executed by a team of threads
}
```

We will go into much more detail in section 6.4.

6.3.3 An actual OpenMP program!

Exercise 6.1 Write a program that contains the following lines:

```
  printf("There are %d processors\n",
         omp_get_num_procs());
#pragma omp parallel
  printf("There are %d threads\n",
        /* !!!! something missing here !!!! */ );
```

The first print statement tells you the number of available cores in the hardware. Your assignment is to supply the missing function that reports the number of threads used. Compile and run the program. Experiment with the `OMP_NUM_THREADS` environment variable. What do you notice about the number of lines printed?

Exercise 6.2 Extend the program from exercise 6.1. Make a complete program based on these lines:

```
int tsum=0;
#pragma omp parallel
  tsum += /* the thread number */
printf("Sum is %d\n",tsum);
```

Compile and run again. (In fact, run your program a number of times.) Do you see something unexpected? Can you think of an explanation?

6.3.4 Code and execution structure

Here are a couple of important concepts:

Definition 6.1structured block An OpenMP directive is followed by an *structured block*; in C this is a single statement, a compound statement, or a block in braces.

A structured block can not be jumped into, so it can not start with a labeled statement, or contain a jump statement leaving the block.

construct An OpenMP *construct* is the section of code starting with a directive and spanning the following structured block.This is a lexical concept: it contains the statements directly enclosed, and not any subroutines called from them.

region of code A *region of code* is defined as all statements that are dynamically encountered while executing the code of an OpenMP construct. This is a dynamic concept: unlike a 'construct', it does include any subroutines that are called from the code in the structured block.

6.4 Parallel Regions

The simplest way to create parallelism in OpenMP is to use the *parallel* pragma. A block preceded by the `omp parallel` pragma is called a *parallel region*; it is executed by a newly created team of threads. This is an instance of the *SPMD* model: all threads execute the same segment of code.

```
#pragma omp parallel
{
  // this is executed by a team of threads
}
```

It would be pointless to have the block be executed identically by all threads. One way to get a meaningful parallel code is to use the `omp_get_thread_num` function, to find out which thread you are, and execute work that is individual to that thread. There is also a function `omp_get_num_threads` to find out the total number of threads. Both these functions give a number relative to the current team; recall from figure 5.1 that new teams can be created recursively.

For instance, if you program computes

```
result = f(x)+g(x)+h(x)
```

you could parallelize this as

```
double result,fresult,gresult,hresult;
#pragma omp parallel
{
  int num = omp_get_thread_num();
  if (num==0)      fresult = f(x);
  else if (num==1) gresult = g(x);
  else if (num==2) hresult = h(x);
}
result = fresult + gresult + hresult;
```

The first thing we want to do is create a team of threads. This is done with a *parallel region*. Here is a very simple example:

```
// hello.c
#pragma omp parallel
  {
    int t = omp_get_thread_num();
    printf("Hello world from %d!\n",t);
  }
```

This code corresponds to the model we just discussed:

- Immediately preceding the parallel block, one thread will be executing the code. In the main program this is the *initial thread*.
- At the start of the block, a new *team of* threads is created, and the thread that was active before the block becomes the *master thread* of that team.
- After the block only the master thread is active.
- Inside the block there is team of threads: each thread in the team executes the body of the block, and it will have access to all variables of the surrounding environment. How many threads there are can be determined in a number of ways; we will get to that later.

Exercise 6.3 Make a full program based on this fragment. Insert different print statements before, inside, and after the parallel region. Run this example. How many times is each print statement executed?

You see that the `parallel` directive

- Is preceded by a special marker: a `#pragma omp` for C/C++;
- Is followed by a single statement or a block in C/C++.

Directives look like *cpp directives*, but they are actually handled by the compiler, not the preprocessor.

Exercise 6.4 Take the 'hello world' program above, and modify it so that you get multiple messages to you screen, saying

```
Hello from thread 0 out of 4!
Hello from thread 1 out of 4!
```

and so on. (The messages may very well appear out of sequence.)

What happens if you set your number of threads larger than the available cores on your computer?

Exercise 6.5 What happens if you call `omp_get_thread_num` and `omp_get_num_threads` outside a parallel region?

6.4.1 Nested parallelism

What happens if you call a function from inside a parallel region, and that function itself contains a parallel region?

```
int main() {
```

```
  ...
#pragma omp parallel
  {
  ...
  func(...)
  ...
  }
} // end of main
void func(...) {
#pragma omp parallel
  {
  ...
  }
}
```

By default, the nested parallel region will have only one thread. To allow nested thread creation, set

```
OMP_NESTED=true
 or
omp_set_nested(1)
```

Exercise 6.6 Test nested parallelism by writing an OpenMP program as follows:

1. Write a subprogram that contains a parallel region.
2. Write a main program with a parallel region; call the subprogram both inside and outside the parallel region.
3. Insert print statements
   (a) in the main program outside the parallel region,
   (b) in the parallel region in the main program,
   (c) in the subprogram outside the parallel region,
   (d) in the parallel region inside the subprogram.

Run your program and count how many print statements of each type you get.

Writing subprograms that are called in a parallel region illustrates the following point: directives are evaluated with respect to the *dynamic scope* of the parallel region, not just the lexical scope. In the following example:

```
#pragma omp parallel
{
  f();
}
void f() {
#pragma omp for
  for ( .... ) {
    ...
```

```
  }
}
```

the body of the function `f` falls in the dynamic scope of the parallel region, so the for loop will be parallelized.

If the function may be called both from inside and outside parallel regions, you can test which is the case with `omp_in_parallel`.

The amount of nested parallelism can be set:

```
OMP_NUM_THREADS=4,2
```

means that initially a parallel region will have four threads, and each thread can create two more threads.

6.5 Loop parallelism

Loop parallelism is a very common type of parallelism in scientific codes, so OpenMP has an easy mechanism for it. OpenMP parallel loops are a first example of OpenMP 'worksharing' constructs (see section 6.6 for the full list): constructs that take an amount of work and distribute it over the available threads in a parallel region.

The parallel execution of a loop can be handled a number of different ways. For instance, you can create a parallel region around the loop, and adjust the loop bounds:

```
#pragma omp parallel
{
  int threadnum = omp_get_thread_num(),
    numthreads = omp_get_num_threads();
  int low = N*threadnum/numthreads,
    high = N*(threadnum+1)/numthreads;
  for (i=low; i<high; i++)
    // do something with i
}
```

A more natural option is to use the *parallel for* pragma:

```
#pragma omp parallel
#pragma omp for
for (i=0; i<N; i++) {
  // do something with i
}
```

This has several advantages. For one, you don't have to calculate the loop bounds for the threads yourself, but you can also tell OpenMP to assign the loop iterations according to different schedules (section 6.5.1).

Figure 6.2 shows the execution on four threads of

```
#pragma omp parallel
{
```

```
  code1();
#pragma omp for
  for (i=1; i<=4*N; i++) {
    code2();
  }
  code3();
}
```

The code before and after the loop is executed identically in each thread; the loop iterations are spread over the four threads.

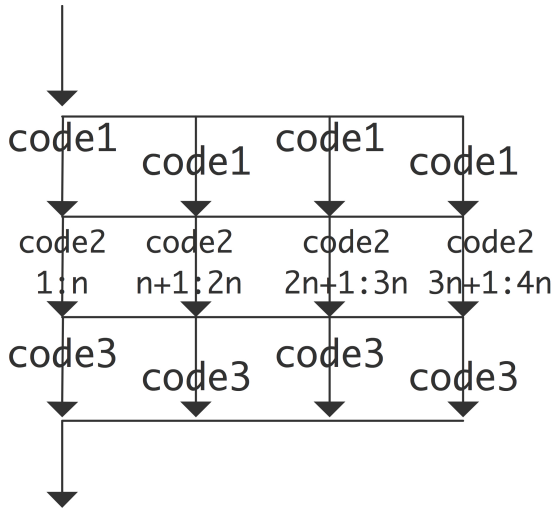


Figure 6.2 Execution of parallel code inside and outside a loop

Note that the `parallel do` and `parallel for` pragmas do not create a team of threads: they take the team of threads that is active, and divide the loop iterations over them.

This means that the `omp for` or `omp do` directive needs to be inside a parallel region. It is also possible to have a combined `omp parallel for` or `omp parallel do` directive.

If your parallel region only contains a loop, you can combine the pragmas for the parallel region and distribution of the loop iterations:

```
#pragma omp parallel for
  for (i=0; .....
```

Exercise 6.7 Compute π by *numerical integration*. We use the fact that π is the area of the unit circle, and we approximate this by computing the area of a quarter circle using *Riemann sums*.

- Let $f(x) = \sqrt{1-x^2}$ be the function that describes the quarter circle for $x = 0 \ldots 1$;

- Then we compute

$$\pi/4 \approx \sum_{i=0}^{N-1} \Delta x f(x_i) \qquad \text{where } x_i = i\Delta x \text{ and } \Delta x = 1/N$$

Write a program for this, and parallelize it using OpenMP parallel for directives.

1. Put a `parallel` directive around your loop. Does it still compute the right result? Does the time go down with the number of threads? (The answers should be no and no.)
2. Change the `parallel` to `parallel for` (or `parallel do`). Now is the result correct? Does execution speed up? (The answers should now be no and yes.)
3. Put a `critical` directive in front of the update. (Yes and very much no.)
4. Remove the `critical` and add a clause`reduction(+:quarterpi)` to the `for` directive. Now it should be correct and efficient.

Use different numbers of cores and compute the speedup you attain over the sequential computation. Is there a performance difference between the OpenMP code with 1 thread and the sequential code?

There are some restrictions on the loop: basically, OpenMP needs to be able to determine in advance how many iterations there will be.

- The loop can not contain `break`, `return`, `exit` statements, or `goto` to a label outside the loop.
- The `continue` (C) statement is allowed.
- The index update has to be an increment (or decrement) by a fixed amount.
- The loop index variable is automatically private, and not changes to it inside the loop are allowed.

6.5.1 Loop schedules

Usually you will have many more iterations in a loop than there are threads. Thus, there are several ways you can assign your loop iterations to the threads. OpenMP lets you specify this with the *schedule* clause.

```
#pragma omp for schedule(....)
```

The first distinction we now have to make is between static and dynamic schedules. With static schedules, the iterations are assigned purely based on the number of iterations and the number of threads (and the `chunk` parameter; see later). In dynamic schedules, on the other hand, iterations are assigned to threads that are unoccupied. Dynamic schedules are a good idea if iterations take an unpredictable amount of time, so that *load balancing* is needed.

Figure 6.3 illustrates this: assume that each core gets assigned two (blocks of) iterations and these blocks take gradually less and less time. You see from the left picture that thread 1 gets two fairly long blocks, where as thread 4 gets

Figure 6.3 Illustration static round-robin scheduling versus dynamic

two short blocks, thus finishing much earlier. (This phenomenon of threads having unequal amounts of work is known as *load imbalance*.) On the other hand, in the right figure thread 4 gets block 5, since it finishes the first set of blocks early. The effect is a perfect load balancing.

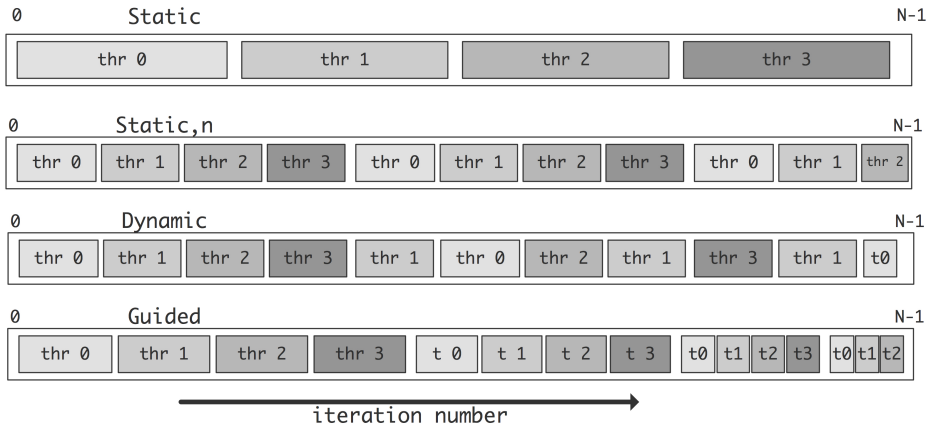


Figure 6.4 Illustration of the scheduling strategies of loop iterations

The default static schedule is to assign one consecutive block of iterations to each thread. If you want different sized blocks you can defined a *schedule chunk* size:

```
#pragma omp for schedule(static[,chunk])
```

(where the square brackets indicate an optional argument). With static scheduling, the compiler will split up the loop iterations at compile time, so, provided the iterations take roughly the same amount of time, this is the most efficient at runtime.

The choice of a chunk size is often a balance between the low overhead of having only a few chunks, versus the load balancing effect of having smaller chunks.

In dynamic scheduling OpenMP will put blocks of iterations (the default chunk size is 1) in a task queue, and the threads take one of these tasks whenever they are finished with the previous.

```
#pragma omp for schedule(static[,chunk])
```

While this schedule may give good load balancing if the iterations take very differing amounts of time to execute, it does carry runtime overhead for managing

the queue of iteration tasks.

Finally, there is the *schedule guided* schedule, which gradually decreases the chunk size. The thinking here is that large chunks carry the least overhead, but smaller chunks are better for load balancing. The various schedules are illustrated in figure 6.4.

If you don't want to decide on a schedule in your code, you can specify the *schedule*runtime schedule. The actual schedule will then at runtime be read from the `OMP_SCHEDULE` environment variable. You can even just leave it to the runtime library by specifying *schedule auto*

Exercise 6.8 We continue with exercise 6.7. We add 'adaptive integration': where needed, the program refines the step size[a]. This means that the iterations no longer take a predictable amount of time.

```
for (i=0; i<nsteps; i++) {
 double
  x = i*h,x2 = (i+1)*h,
  y = sqrt(1-x*x),y2 = sqrt(1-x2*x2),
  slope = (y-y2)/h;
 if (slope>15) slope = 15;
 int
  samples = 1+(int)slope, is;
 for (is=0; is<samples; is++) {
  double
     hs = h/samples,
     xs = x+ is*hs,
     ys = sqrt(1-xs*xs);
  quarterpi += hs*ys;
  nsamples++;
 }
}
pi = 4*quarterpi;
```

1. Use the `omp parallel for` construct to parallelize the loop. As in the previous lab, you may at first see an incorrect result. Use the `reduction` clause to fix this.
2. Your code should now see a decent speedup, using up to 8 cores. However, it is possible to get completely linear speedup. For this you need to adjust the schedule.
   Start by using `schedule(static,$n$)`. Experiment with values for n. When can you get a better speedup? Explain this.
3. Since this code is somewhat dynamic, try `schedule(dynamic)`. This will

actually give a fairly bad result. Why? Use `schedule(dynamic,$n$)` instead, and experiment with values for n.
4. Finally, use `schedule(guided)`, where OpenMP uses a heuristic. What results does that give?

[a] It doesn't actually do this in a mathematically sophisticated way, so this code is more for the sake of the example.

Exercise 6.9 Program the *LU factorization* algorithm without pivoting.

```
for k=1,n:
  A[k,k] = 1./A[k,k]
  for i=k+1,n:
    A[i,k] = A[i,k]/A[k,k]
    for j=k+1,n:
      A[i,j] = A[i,j] - A[i,k]*A[k,j]
```

1. Argue that it is not possible to parallelize the outer loop.
2. Argue that it is possible to parallelize both the i and j loops.
3. Parallelize the algorithm by focusing on the i loop. Why is the algorithm as given here best for a matrix on row-storage? What would you do if the matrix was on column storage?
4. Argue that with the default schedule, if a row is updated by one thread in one iteration, it may very well be updated by another thread in another. Can you find a way to schedule loop iterations so that this does not happen? What practical reason is there for doing so?

The schedule can be declared explicitly, set at runtime through the environment variable `OMP_SCHEDULE`, or left up to the runtime system by specifying `auto`. Especially in the last two cases you may want to enquire what schedule is currently being used with `omp_get_schedule`.

```
int omp_get_schedule(omp_sched_t * kind, int * modifier );
```

Its mirror call is `omp_set_schedule`, which sets the value that is used when schedule value `runtime` is used. It is in effect equivalent to setting the environment variable `OMP_SCHEDULE`.

```
void omp_set_schedule (omp_sched_t kind, int modifier);
```

Type	environment variable `OMP_SCHEDULE=`	clause `schedule( ... )`	modifier default
static	static[,n]	static[,n]	$N/nthreads$
dynamic	dynamic[,n]	dynamic[,n]	1
guided	guided[,n]	guided[,n]	

Here are the various schedules you can set with the `schedule` clause:

affinity Set by using value `omp_sched_affinity`;

auto The schedule is left up to the implementation. Set by using value `omp_sched_auto`;

dynamic value: 2. The modifier parameter is the *chunk* size; default 1. Set by using value `omp_sched_dynamic`;

guided Value: 3. The modifier parameter is the *chunk* size. Set by using value `omp_sched_guided`;

runtime Use the value of the `OMP_SCHEDULE` environment variable. Set by using value `omp_sched_runtime`;

static value: 1. The modifier parameter is the *chunk* size. Set by using value `omp_sched_static`.

6.5.2 Collapsing nested loops

In general, the more work there is to divide over a number of threads, the more efficient the parallelization will be. In the context of parallel loops, it is possible to increase the amount of work by parallelizing all levels of loops instead of just the outer one.

Example: in

```
for ( i=0; i<N; i++ )
  for ( j=0; j<N; j++ )
    A[i][j] = B[i][j] + C[i][j]
```

all N^2 iterations are independent, but a regular `omp for` directive will only parallelize one level. The *collapse* clause will parallelize more than one level:

```
#pragma omp for collapse(2)
for ( i=0; i<N; i++ )
  for ( j=0; j<N; j++ )
    A[i][j] = B[i][j] + C[i][j]
```

It is only possible to collapse perfectly nested loops, that is, the loop body of the outer loop can consist only of the inner loop; there can be no statements before or after the inner loop in the loop body of the outer loop. That is, the two loops in

```
for (i=0; i<N; i++) {
  y[i] = 0.;
  for (j=0; j<N; j++)
    y[i] += A[i][j] * x[j]
  }
```

can not be collapsed.

6.5.3 Ordered iterations

Iterations in a parallel loop that are execution in parallel do not execute in lockstep. That means that in

```
#pragma omp parallel for
for ( ... i ... ) {
  ... f(i) ...
  printf("something with %d\n",i);
}
```

it is not true that all function evaluations happen more or less at the same time, followed by all print statements. The print statements can really happen in any order. The *ordered* clause coupled with the *ordered* directive can force execution in the right order:

```
#pragma omp parallel for ordered
for ( ... i ... ) {
  ... f(i) ...
#pragma omp ordered
  printf("something with %d\n",i);
}
```

Example code structure:

```
#pragma omp parallel for shared(y) ordered
for ( ... i ... ) {
  int x = f(i)
#pragma omp ordered
  y[i] += f(x)
  z[i] = g(y[i])
}
```

There is a limitation: each iteration can encounter only one `ordered` directive.

6.5.4 nowait

The implicit barrier at the end of a work sharing construct can be cancelled with a `nowait` clause. This has the effect that threads that are finished can continue with the next code in the parallel region:

```
#pragma omp parallel
{
#pragma omp for nowait
  for (i=0; i<N; i++) { ... }
  // more parallel code
}
```

In the following example, threads that are finished with the first loop can start on the second. Note that this requires both loops to have the same schedule. We specify the static schedule here to have an identical scheduling of iterations over threads:

```
#pragma omp parallel
{
  x = local_computation()
#pragma omp for schedule(static) nowait
  for (i=0; i<N; i++) {
    x[i] = ...
  }
#pragma omp for schedule(static)
```

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

6.5.5 While loops

OpenMP can only handle 'for' loops: *while loops* can not be parallelized. So you have to find a way around that. While loops are for instance used to search through data:

```
while ( a[i]!=0 && i<imax ) {
 i++; }
// now i is the first index for which \cpp{a[i]} is zero.
```

We replace the while loop by a for loop that examines all locations:

```
result = -1;
#pragma omp parallel for
for (i=0; i<imax; i++) {
  if (a[i]!=0 && result<0) result = i;
}
```

Exercise 6.10 Show that this code has a race condition.

You can fix the race condition by making the condition into a critical section; section 6.9.2. In this particular example, with a very small amount of work per iteration, that is likely to be inefficient in this case (why?). A more efficient solution uses the *lastprivate* pragma:

```
result = -1;
#pragma omp parallel for lastprivate(result)
for (i=0; i<imax; i++) {
  if (a[i]!=0) result = i;
}
```

You have now solved a slightly different problem: the result variable contains the *last* location where `a[i]` is zero.

6.6 Work Sharing

The declaration of a *parallel region* establishes a team of threads. This offers the possibility of parallelism, but to actually get meaningful parallel activity you need something more. OpenMP uses the concept of a *work sharing construct*: a way of dividing parallelizable work over a team of threads. The work sharing constructs are:

- `for`. The threads divide up the loop iterations among themselves; see 6.5.
- `sections` The threads divide a fixed number of sections between themselves; see section 6.6.1.

- `single` The section is executed by a single thread; section 6.6.2.
- `task` out of scope of this reader.

6.6.1 Sections

A parallel loop is an example of independent work units that are numbered. If you have a pre-determined number of independent work units, the `sections` is more appropriate. In a `sections` construct can be any number of `section` constructs. These need to be independent, and they can be execute by any available thread in the current team, including having multiple sections done by the same thread.

```
#pragma omp sections
{
#pragma omp section
  // one calculation
#pragma omp section
  // another calculation
}
```

This construct can be used to divide large blocks of independent work. Suppose that in the following line, both `f(x)` and `g(x)` are big calculations:

```
y = f(x) + g(x)
```

You could then write

```
double y1,y2;
#pragma omp sections
{
#pragma omp section
  y1 = f(x)
#pragma omp section
  y2 = g(x)
}
y = y1+y2;
```

Instead of using two temporaries, you could also use a critical section; see section 6.9.2. However, the best solution is have a `reduction` clause on the `sections` directive:

```
y = f(x) + g(x)
```

You could then write

```
y = 0;
#pragma omp sections reduction(+:y)
{
#pragma omp section
  y += f(x)
#pragma omp section
```

```
  y += g(x)
}
```

6.6.2 Single/master

The `single` and `master` pragma limit the execution of a block to a single thread. This can for instance be used to print tracing information or doing I/O operations.

```
#pragma omp parallel
{
#pragma omp single
  printf("We are starting this section!\n");
  // parallel stuff
}
```

Another use of `single` is to perform initializations in a parallel region:

```
int a;
#pragma omp parallel
{
  #pragma omp single
    a = f(); // some computation
  #pragma omp sections
    // various different computations using a
}
```

The point of the single directive in this last example is that the computation needs to be done only once, because of the shared memory. Since it's a work sharing construct there is an *implicit barrier* after it, which guarantees that all threads have the correct value in their local memory.

The `master` directive, also enforces execution on a single thread, specifically the master thread of the team, but it does not have the synchronization through the implicit barrier.

Exercise 6.11 Modify the above code to read:

```
int a;
#pragma omp parallel
{
  #pragma omp master
    a = f(); // some computation
  #pragma omp sections
    // various different computations using a
}
```

This code is no longer correct. Explain.

Above we motivated the `single` directive as a way of initializing shared variables. It is also possible to use `single` to initialize private variables. In

that case you add the `copyprivate` clause. This is a good solution if setting the variable takes I/O.

6.7 Controlling Thread Data

In a parallel region there are two types of data: private and shared. In this sections we will see the various way you can control what category your data falls under; for private data items we also discuss how their values relate to shared data.

6.7.1 Shared data

In a parallel region, any data declared outside it will be shared: any thread using a variable `x` will access the same memory location associated with that variable.

Example:

```
  int x = 5;
#pragma omp parallel
  {
    x = x+1;
    printf("shared: x is %d\n",x);
  }
```

All threads increment the same variable, so after the loop it will have a value of five plus the number of threads; or maybe less because of the data races involved. See 5.1.5 for an explanation of the issues involved; see 6.9.2 for a solution in OpenMP.

Sometimes this global update is what you want; in other cases the variable is intended only for intermediate results in a computation. In that case there are various ways of creating data that is local to a thread, and therefore invisible to other threads.

6.7.2 Private data

In the C/C++ language it is possible to declare variables inside a *lexical scope*; roughly: inside curly braces. This concept extends to OpenMP parallel regions and directives: any variable declared in a block following an OpenMP directive will be local to the executing thread.

Example:

```
  int x = 5;
#pragma omp parallel
  {
    int x; x = 3;
    printf("local: x is %d\n",x);
  }
```

After the parallel region the outer variable `x` will still have the value `5`: there is no *storage association* between the private variable and global one.

The **private** directive declares data to have a separate copy in the memory of each thread. Such private variables are initialized as they would be in a main program. Any computed value goes away at the end of the parallel region. (However, see below.) Thus, you should not rely on any initial value, or on the value of the outer variable after the region.

```
  int x = 5;
#pragma omp parallel private(x)
  {
    x = x+1; // dangerous
    printf("private: x is %d\n",x);
  }
  printf("after: x is %d\n",x); // also dangerous
```

Data that is declared private with the **private** directive is put on a separate *stack per thread*. The OpenMP standard does not dictate the size of these stacks, but beware of *stack overflow*. A typical default is a few megabyte; you can control it with the environment variable `OMP_STACKSIZE`. Its values can be literal or with suffixes: 123 456k 567K 678m 789M 246g 357G

A normal *Unix process* also has a stack, but this is independent of the OpenMP stacks for private data. You can query or set the Unix stack with `ulimit`: [] ulimit -s 64000 [] ulimit -s 8192 [] ulimit -s 8192 The Unix stack can grow dynamically as space is needed. This does not hold for the OpenMP stacks: they are immediately allocated at their requested size. Thus it is important not too make them too large.

6.7.3 Data in dynamic scope

Functions that are called from a parallel region fall in the *dynamic scope* of that parallel region. The rules for variables in that function are as follows:

- Any variables locally defined to the function are private.
- **static** variables in C are shared.
- The function arguments inherit their status from the calling environment.

6.7.4 Temporary variables in a loop

It is common to have a variable that is set and used in each loop iteration:

```
#pragma omp parallel for
for ( ... i ... ) {
  x = i*h;
  s = sin(x); c = cos(x);
  a[i] = s+c;
  b[i] = s-c;
}
```

By the above rules, the variables `x,s,c` are all shared variables. However, the values they receive in one iteration are not used in a next iteration, so they behave in fact like private variables to each iteration.

- In both C and Fortran you can declare these variables private in the parallel for directive.
- In C, you can also redefine the variables inside the loop.

Sometimes, even if you forget to declare these temporaries as private, the code may still give the correct output. That is because the compiler can sometimes eliminate them from the loop body, since it detects that their values are not otherwise used.

6.7.5 Default

- Loop variables in an `omp for` are private;
- Local variables in the parallel region are private.

You can alter this default behaviour with the *default* clause:

```
#pragma omp parallel default(shared) private(x)
{ ... }
#pragma omp parallel default(private) shared(matrix)
{ ... }
```

and if you want to play it safe:

```
#pragma omp parallel default(none) private(x) shared(matrix)
{ ... }
```

- The *default shared* clause means that all variables from the outer scope are shared in the parallel region; any private variables need to be declared explicitly. This is the default behaviour.
- The *default private* clause means that all outer variables become private in the parallel region. They are not initialized; see the next option. Any shared variables in the parallel region need to be declared explicitly. This value is not available in C.
- The *default firstprivate* clause means all outer variables are private in the parallel region, and initialized with their outer value. Any shared variables need to be declared explicitly. This value is not available in C.
- The *default none* option is good for debugging, because it forces you to specify for each variable in the parallel region whether it's private or shared. Also, if your code behaves differently in parallel from sequential there is probably a data race. Specifying the status of every variable is a good way to debug this.

6.7.6 Array data

The rules for arrays are slightly different from those for scalar data:

1. Statically allocated data, that is with a syntax like

   ```
   int array[100];
   integer,dimension(:) :: array(100)
   ```

   can be shared or private, depending on the clause you use.
2. Dynamically allocated data, that is, created with `malloc` or `allocate`, can only be shared.

Example of the first type: in

```
// alloc3.c
int array[nthreads];
{
  int t = 2;
  array += t;
  array[0] = t;
}
```

each thread gets a private copy of the array, properly initialized.

On the other hand, in

```
// alloc1.c
int *array = (int*) malloc(nthreads*sizeof(int));
#pragma omp parallel firstprivate(array)
{
  int t = omp_get_thread_num();
  array += t;
  array[0] = t;
}
```

each thread gets a private pointer, but all pointers point to the same object.

6.7.7 First and last private

Above, you saw that private variables are completely separate from any variables by the same name in the surrounding scope. However, there are two cases where you may want some *storage association* between a private variable and a global counterpart.

First of all, private variables are created with an undefined value. You can force their initialization with `firstprivate`.

```
  int t=2;
#pragma omp parallel firstprivate(t)
  {
    t += f( omp_get_thread_num() );
    g(t);
  }
```

The variable `t` behaves like a private variable, except that it is initialized to the outside value.

Secondly, you may want a private value to be preserved to the environment outside the parallel region. This really only makes sense in one case, where you preserve a private variable from the last iteration of a parallel loop, or the last section in an `sections` construct. This is done with `lastprivate`:

```
#pragma omp parallel for \
        lastprivate(tmp)
for (i=0; i<N; i+) {
```

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

6.7.8 Persistent data through `threadprivate`

Most data in OpenMP parallel regions is either inherited from the master thread and therefore shared, or temporary within the scope of the region and fully private. There is also a mechanism for *thread-private data*, which is not limited in lifetime to one parallel region. The `threadprivate` pragma is used to declare that each thread is to have a private copy of a variable:

```
#pragma omp threadprivate(var)
```

The variable needs be:

- a file or static variable in C, or
- a static class member in C++.

Thread private initialization

If each thread needs a different value in its threadprivate variable, the initialization needs to happen in a parallel region.

In the following example a team of 7 threads is created, all of which set their thread-private variable. Later, this variable is read by a larger team: the variables that have not been set are undefined, though often simply zero:

```
// threadprivate.c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

static int tp;

int main(int argc,char **argv) {

#pragma omp threadprivate(tp)

#pragma omp parallel num_threads(7)
  tp = omp_get_thread_num();

#pragma omp parallel num_threads(9)
  printf("Thread %d has %d\n",omp_get_thread_num(),tp);

  return 0;
}
```

On the other hand, if the thread private data starts out identical in all threads, the `copyin` clause can be used:

```
#pragma omp threadprivate(private_var)

private_var = 1;
#pragma omp parallel copyin(private_var)
  private_var += omp_get_thread_num()
```

If one thread needs to set all thread private data to its value, the `copyprivate` clause can be used:

```
#pragma omp parallel
{
  ...
#pragma omp single copyprivate(private_var)
  private_var = read_data();
  ...
}
```

Thread private example

The typical application for thread-private variables is in *random number generation*. A random number generator needs saved state, since it computes each next value from the current one. To have a parallel generator, each thread will create and initialize a private 'current value' variable. This will persist even when the execution is not in a parallel region; it gets updated only in a parallel region.

> **Exercise 6.12** Calculate the area of the *Mandelbrot set* by random sampling. Initialize the random number generator separately for each thread; then use a parallel loop to evaluate the points. Explore performance implications of the different loop scheduling strategies.

Threadprivate variables require `OMP_DYNAMIC` to be switched off.

6.8 Reductions

Parallel tasks often produce some quantity that needs to be summed or otherwise combined. In section 6.4 you saw an example, and it was stated that the solution given there was not very good.

The problem in that example was the *race condition* involving the `result` variable. The simplest solution is to eliminate the race condition by declaring a *critical section*:

```
double result = 0;
#pragma omp parallel
{
  double local_result;
  int num = omp_get_thread_num();
  if (num==0)      local_result = f(x);
  else if (num==1) local_result = g(x);
```

```
  else if (num==2) local_result = h(x);
#pragma omp critical
  result += local_result;
}
```

This is a good solution if the amount of serialization in the critical section is small compared to computing the functions f, g, h. On the other hand, you may not want to do that in a loop:

```
double result = 0;
#pragma omp parallel
{
  double local_result;
#pragma omp for
  for (i=0; i<N; i++) {
    local_result = f(x,i);
#pragma omp critical
    result += local_result;
  } // end of for loop
}
```

Exercise 6.13 Can you think of a small modification of this code, that still uses a critical section, that is more efficient? Time both codes.

The easiest way to effect a reduction is of course to use the `reduction` clause. Adding this to an `omp for` or an `omp sections` construct has the following effect:

- OpenMP will make a copy of the reduction variable per thread, initialized to the identity of the reduction operator, for instance 1 for multiplication.
- Each thread will then reduce into its local variable;
- At the end of the loop, the local results are combined, again using the reduction operator, into the global variable.

This is one of those cases where the parallel execution can have a slightly different value from the one that is computed sequentially, because floating point operations are not associative.

If your code can not be easily structure as a reduction, you can realize the above scheme by hand by 'duplicating' the global variable and gather the contributions later. This example presumes three threads, and gives each a location of their own to store the result computed on that thread:

```
double result,local_results[3];
#pragma omp parallel
{
  int num = omp_get_thread_num();
  if (num==0)      local_results[num] = f(x)
  else if (num==1) local_results[num] = g(x)
  else if (num==2) local_results[num] = h(x)
}
```

```
result = local_results[0]+local_results[1]+local_results[2]
```

While this code is correct, it may be inefficient because of a phenomemon called *false sharing*. Even though the threads write to separate variables, those variables are likely to be on the same *cacheline*.This means that the cores will be wasting a lot of time and bandwidth updating each other's copy of this cacheline.

False sharing can be prevent by giving each thread its own cacheline:

```
double result,local_results[3][8];
#pragma omp parallel
{
  int num = omp_get_thread_num();
  if (num==0)      local_results[num][1] = f(x)
// et cetera
}
```

A more elegant solution gives each thread a true local variable, and uses a critical section to sum these, at the very end:

```
double result = 0;
#pragma omp parallel
{
  double local_result;
  local_result = .....
#pragam omp critical
  result += local_result;
}
```

6.8.1 Built-in reduction operators

Arithmetic reductions: $+, *, -, \max, \min$

Logical operator reductions in C: `& && | || ^`

6.8.2 Initial value for reductions

The treatment of initial values in reductions is slightly involved.

```
  x = init_x
#pragma omp parallel for reduction(min:x)
  for (int i=0; i<N; i++)
    x = min(x,data[i]);
```

Each thread does a partial reduction, but its initial value is not the user-supplied `init_x` value, but a value dependent on the operator. In the end, the partial results will then be combined with the user initial value. The initialization values are mostly self-evident, such as zero for addition and one for multiplication. For min and max they are respectively the maximal and minimal representable value of the result type.

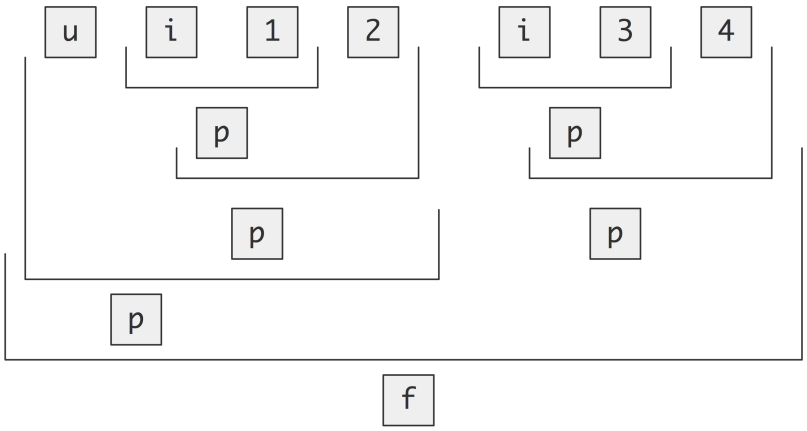


Figure 6.5 Reduction of four items on two threads, taking into account initial values.

Figure 6.5 illustrates this, where `1,2,3,4` are four data items, `i` is the OpenMP initialization, and `u` is the user initialization; each `p` stands for a partial reduction value. The figure is based on execution using two threads.

6.8.3 User-defined reductions

With *user-defined reductions*, the programmer specifies the function that does the elementwise comparison. This takes two steps.

1. You need a function of two arguments that returns the result of the comparison. You can do this yourself, but, especially with the C++ standard library, you can use functions such as `std::vector::insert`.
2. Specifying how this function operates on two variables `omp_out` and `omp_in`, corresponding to the partially reduced result and the new operand respectively. The new partial result should be left in `omp_out`.
3. Optionally, you can specify the value to which the reduction should be initialized.

This is the syntax of the definition of the reduction, which can then be used in multiple `reduction` clauses.

```
#pragma omp declare reduction
    ( identifier : typelist : combiner )
    [initializer(initializer-expression)]
```

where:

`identifier` is a name; this can be overloaded for different types, and redefined in inner scopes.

`typelist` is a list of types.

`combiner` is an expression that updates the internal variable `omp_out` as function of itself and `omp_in`.

`initializer` sets `omp_priv` to the identity of the reduction; this can be an expression or a brace initializer.

For instance, recreating the maximum reduction would look like this:

```
// ireduct.c
int mymax(int r,int n) {
```

```
  // r is the already reduced value
  // n is the new value
  int m;
  if (n>r) {
    m = n;
  } else {
    m = r;
  }
  return m;
}
#pragma omp declare reduction \
  (rwz:int:omp_out=mymax(omp_out,omp_in)) \
  initializer(omp_priv=INT_MIN)
  m = INT_MIN;
#pragma omp parallel for reduction(rwz:m)
  for (int idata=0; idata<ndata; idata++)
    m = mymax(m,data[idata]);
```

Exercise 6.14 Write a reduction routine that operates on an array of non-negative integers, finding the smallest nonzero one. If the array has size zero, or entirely consists of zeros, return -1.

6.8.4 Reductions and floating-point math

The mechanisms that OpenMP uses to make a reduction parallel go against the strict rules for floating point expression evaluation in C. OpenMP ignores this issue: it is the programmer's job to ensure proper rounding behaviour.

6.9 Synchronization

In the constructs for declaring parallel regions above, you had little control over in what order threads executed the work they were assigned. This section will discuss *synchronization* constructs: ways of telling threads to bring a certain order to the sequence in which they do things.

- `critical`: a section of code can only be executed by one thread at a time; see 6.9.2.
- `atomic` Update of a single memory location. Only certain specified syntax pattterns are supported. This was added in order to be able to use hardware support for atomic updates.
- `barrier`: section 6.9.1.
- `ordered`: section 6.5.3.
- locks: section 6.9.3.
- `nowait`: section 6.5.4.

6.9.1 Barrier

A barrier defines a point in the code where all active threads will stop until all threads have arrived at that point. With this, you can guarantee that certain calculations are finished. For instance, in this code snippet, computation of `y` can not proceed until another thread has computed its value of `x`.

```
#pragma omp parallel
{
  int mytid = omp_get_thread_num();
  x[mytid] = some_calculation();
  y[mytid] = x[mytid]+x[mytid+1];
}
```

This can be guaranteed with a `barrier` pragma:

```
#pragma omp parallel
{
  int mytid = omp_get_thread_num();
  x[mytid] = some_calculation();
#pragma omp barrier
  y[mytid] = x[mytid]+x[mytid+1];
}
```

Apart from the barrier directive, which inserts an explicit barrier, OpenMP has *implicit barriers* after a load sharing construct. Thus the following code is well defined:

```
#pragma omp parallel
{
#pragma omp for
  for (int mytid=0; mytid<number_of_threads; mytid++)
    x[mytid] = some_calculation();
#pragma omp for
  for (int mytid=0; mytid<number_of_threads-1; mytid++)
    y[mytid] = x[mytid]+x[mytid+1];
}
```

You can also put each parallel loop in a parallel region of its own, but there is some overhead associated with creating and deleting the team of threads in between the regions.

Implicit barriers

At the end of a parallel region the team of threads is dissolved and only the master thread continues. Therefore, there is an *implicit barrier at the end of a parallel region*.

There is some *barrier behaviour* associated with `omp for` loops and other *worksharing constructs*. For instance, there is an *implicit barrier* at the end of the loop. This barrier behaviour can be cancelled with the `nowait` clause.

You will often see the idiom

```
#pragma omp parallel
{
#pragma omp for nowait
  for (i=0; i<N; i++)
    a[i] = // some expression
#pragma omp for
  for (i=0; i<N; i++)
    b[i] = ..... a[i] .....
```

Here the `nowait` clause implies that threads can start on the second loop while other threads are still working on the first. Since the two loops use the same schedule here, an iteration that uses `a[i]` can indeed rely on it that that value has been computed.

6.9.2 Mutual exclusion

Sometimes it is necessary to let only one thread execute a piece of code. Such a piece of code is called a *critical section*, and OpenMP has several mechanisms for realizing this.

The most common use of critical sections is to update a variable. Since updating involves reading the old value, and writing back the new, this has the possibility for a *race condition*: another thread reads the current value before the first can update it; the second thread the updates to the wrong value.

Critical sections are an easy way to turn an existing code into a correct parallel code. However, there are disadvantages to this, and sometimes a more drastic rewrite is called for.

`critical` and `atomic`

There are two pragmas for critical sections: `critical` and `atomic`. The second one is more limited but has performance advantages.

The typical application of a critical section is to update a variable:

```
#pragma omp parallel
{
  int mytid = omp_get_thread_num();
  double tmp = some_function(mytid);
#pragma omp critical
  sum += tmp;
}
```

Exercise 6.15 Consider a loop where each iteration updates a variable.

```
#pragma omp parallel for shared(result)
  for ( i ) {
      result += some_function_of(i);
  }
```

Discuss qualitatively the difference between:

- turning the update statement into a critical section, versus
- letting the threads accumulate into a private variable `tmp` as above, and summing these after the loop.

Do an Ahmdal-style quantitative analysis of the first case, assuming that you do n iterations on p threads, and each iteration has a critical section that takes a fraction f. Assume the number of iterations n is a multiple of the number of threads p. Also assume the default static distribution of loop iterations over the threads.

A `critical` section works by acquiring a lock, which carries a substantial overhead. Furthermore, if your code has multiple critical sections, they are all mutually exclusive: if a thread is in one critical section, the other ones are all blocked.

On the other hand, the syntax for `atomic` sections is limited to the update of a single memory location, but such sections are not exclusive and they can be more efficient, since they assume that there is a hardware mechanism for making them critical.

The problem with `critical` sections being mutually exclusive can be mitigated by naming them:

```
#pragma omp critical (optional_name_in_parens)
```

6.9.3 Locks

OpenMP also has the traditional mechanism of a *lock*. A lock is somewhat similar to a critical section: it guarantees that some instructions can only be performed by one process at a time. However, a critical section is indeed about code; a lock is about data. With a lock you make sure that some data elements can only be touched by one process at a time.

One simple example of the use of locks is generation of a *histogram*. A histogram consists of a number of bins, that get updated depending on some data. Here is the basic structure of such a code:

```
int count[100];
float x = some_function();
int ix = (int)x;
if (ix>=100)
  error();
else
  count[ix]++;
```

It would be possible to guard the last line:

```
#pragma omp critical
  count[ix]++;
```

but that is unnecessarily restrictive. If there are enough bins in the histogram, and if the `some_function` takes enough time, there are unlikely to be conflicting writes. The solution then is to create an array of locks, with one lock for each `count` location.

Create/destroy:

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

Set and release:

```
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
```

Since the set call is blocking, there is also

```
omp_test_lock();
```

Unsetting a lock needs to be done by the thread that set it.
Lock operations implicitly have a `flush`.

Exercise 6.16 In the following code, one process sets array A and then uses it to update B; the other process sets array B and then uses it to update A. Argue that this code can deadlock. How could you fix this?

```
#pragma omp parallel shared(a,b,nthreads,locka,lockb)
  #pragma omp sections nowait
    {
    #pragma omp section
      {
      omp_set_lock(&locka);
      for (i=0; i<N; i++)
        a[i] = ..

      omp_set_lock(&lockb);
      for (i=0; i<N; i++)
        b[i] = .. a[i] ..
      omp_unset_lock(&lockb);
      omp_unset_lock(&locka);
      }

    #pragma omp section
      {
      omp_set_lock(&lockb);
      for (i=0; i<N; i++)
        b[i] = ...

      omp_set_lock(&locka);
      for (i=0; i<N; i++)
        a[i] = .. b[i] ..
      omp_unset_lock(&locka);
      omp_unset_lock(&lockb);
      }
    }  /* end of sections */
  }  /* end of parallel region */
```

Nested locks

A lock as explained above can not be locked if it is already locked. A *nested lock* can be locked multiple times by the same thread before being unlocked.

- `omp_init_nest_lock`
- `omp_destroy_nest_lock`
- `omp_set_nest_lock`
- `omp_unset_nest_lock`
- `omp_test_nest_lock`

6.9.4 Example: Fibonacci computation

The *Fibonacci sequence* is recursively defined as

$$F(0) = 1, \qquad F(1) = 1, \qquad F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2.$$

We start by sketching the basic single-threaded solution. The naive code looks like:

```
int main() {
  value = new int[nmax+1];
  value[0] = 1;
  value[1] = 1;
  fib(10);
}

int fib(int n) {
  int i, j, result;
  if (n>=2) {
    i=fib(n-1); j=fib(n-2);
    value[n] = i+j;
  }
  return value[n];
}
```

Howver, this is inefficienty, since most intermediate values will be computed more than once. We solve this by keeping track of which results are known:

```
  ...
  done = new int[nmax+1];
  for (i=0; i<=nmax; i++)
    done[i] = 0;
  done[0] = 1;
  done[1] = 1;
  ...
int fib(int n) {
  int i, j;
  if (!done[n]) {
    i = fib(n-1); j = fib(n-2);
    value[n] = i+j; done[n] = 1;
  }
  return value[n];
}
```

The OpenMP parallel solution calls for two different ideas. First of all, we parallelize the recursion by using tasks:

```
int fib(int n) {
  int i, j;
  if (n>=2) {
```

```
#pragma omp task shared(i) firstprivate(n)
    i=fib(n−1);
#pragma omp task shared(j) firstprivate(n)
    j=fib(n−2);
#pragma omp taskwait
    value[n] = i+j;
  }
  return value[n];
}
```

This computes the right solution, but, as in the naive single-threaded solution, it recomputes many of the intermediate values.

A naive addition of the `done` array leads to data races, and probably an incorrect solution:

```
int fib(int n) {
  int i, j, result;
  if (!done[n]) {
#pragma omp task shared(i) firstprivate(n)
    i=fib(n−1);
#pragma omp task shared(i) firstprivate(n)
    j=fib(n−2);
#pragma omp taskwait
    value[n] = i+j;
    done[n] = 1;
  }
  return value[n];
}
```

For instance, there is no guarantee that the `done` array is updated later than the `value` array, so a thread can think that `done[n−1]` is true, but `value[n−1]` does not have the right value yet.

One solution to this problem is to use a lock, and make sure that, for a given index `n`, the values `done[n]` and `value[n]` are never touched by more than one thread at a time:

```
int fib(int n)
{
  int i, j;
  omp_set_lock( &(dolock[n]) );
  if (!done[n]) {
#pragma omp task shared(i) firstprivate(n)
    i = fib(n−1);
#pragma omp task shared(j) firstprivate(n)
    j = fib(n−2);
#pragma omp taskwait
    value[n] = i+j;
    done[n] = 1;
```

```
  }
  omp_unset_lock( &(dolock[n]) );
  return value[n];
}
```

This solution is correct, optimally efficient in the sense that it does not recompute anything, and it uses tasks to obtain a parallel execution.

However, the efficiency of this solution is only up to a constant. A lock is still being set, even if a value is already computed and therefore will only be read. This can be solved with a complicated use of critical sections, but we will forego this.

PART IV
MESSAGE PASSING INTERFACE

7. Message Passing Interface (MPI)

The following chapter contains reference material for using the Message Passing Interface (MPI) in parallel programming. This chapter is an extension of section 5.3.3, and intended as reference only.

7.1 Getting started with MPI

In this chapter you will learn the use of the main tool for distributed memory programming: the *Message Passing Interface* (MPI) library The MPI library has about 250 routines, many of which you may never need. Since this is a textbook, not a complete reference manual, we will focus on the important concepts and give the important routines for each concept. What you learn here should be enough for most common purposes. You are advised to keep a reference document handy, in case there is a specialized routine, or to look up subtleties about the routines you use.

7.1.1 Distributed memory and message passing

In its simplest form, a distributed memory machine is a collection of single computers hooked up with network cables. In fact, this has a name: a *Beowulf cluster*. As you recognize from that setup, each processor can run an independent program, and has its own memory without direct access to other processors' memory. MPI is the magic that makes multiple instantiations of the same executable run so that they know about each other and can exchange data through the network.

One of the reasons that MPI is so successful as a tool for high performance on clusters is that it is very explicit: the programmer controls many details of the data motion between the processors. Consequently, a capable programmer can write very efficient code with MPI. Unfortunately, that programmer will

have to spell things out in considerable detail. For this reason, people sometimes call MPI 'the assembly language of parallel programming'. If that sounds scary, be assured that things are not that bad. You can get started fairly quickly with MPI, using just the basics, and coming to the more sophisticated tools only when necessary.

Another reason that MPI was a big hit with programmers is that it does not ask you to learn a new language: it is a library that can be interface to C/C++ or Fortran; there are even bindings to Python. A related point is that it is easy to install: there are free implementations that you can download and install on any computer that has a Unix-like operating system, even if that is not a parallel machine.

7.1.2 History

Before the MPI standard was developed in 1993-4, there were many libraries for distributed memory computing, often proprietary to a vendor platform. MPI standardized the inter-process communication mechanisms. Other features, such as process management in *PVM*, or parallel I/O were omitted. Later versions of the standard have included many of these features.

Since MPI was designed by a large number of academic and commercial participants, it quickly became a standard. A few packages from the pre-MPI era, such as *Charmpp* [30], are still in use since they support mechanisms that do not exist in MPI.

7.1.3 Basic model

Here we sketch the two most common scenarios for using MPI. In the first, the user is working on an interactive machine, which has network access to a number of hosts, typically a network of workstations; see figure 7.1. The user

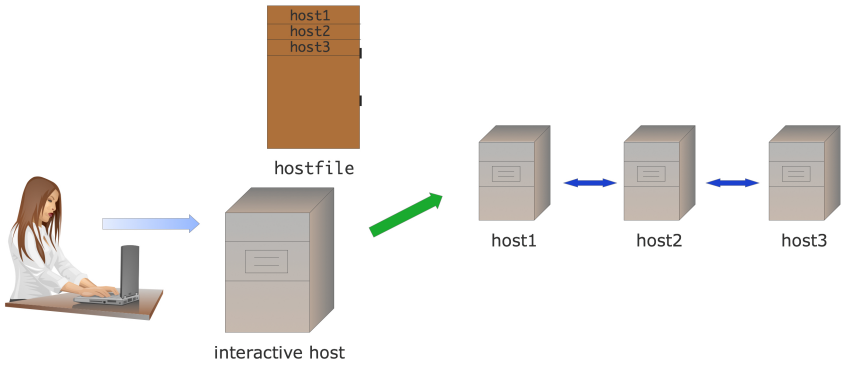


Figure 7.1 Interactive MPI setup

types the command `mpiexec`[12] and supplies

- The number of hosts involved,
- their names, possibly in a hostfile,

[12] A command variant is `mpirun`; your local cluster may have a different mechanism.

- and other parameters, such as whether to include the interactive host; followed by
- the name of the program and its parameters.

The `mpirun` program then makes an `ssh` connection to each of the hosts, giving them sufficient information that they can find each other. All the output of the processors is piped through the `mpirun` program, and appears on the interactive console.

In the second scenario (figure 7.2) the user prepares a *batch job* script with commands, and these will be run when the *batch scheduler* gives a number of hosts to the job. Now the batch script contains the `mpirun` command, and the

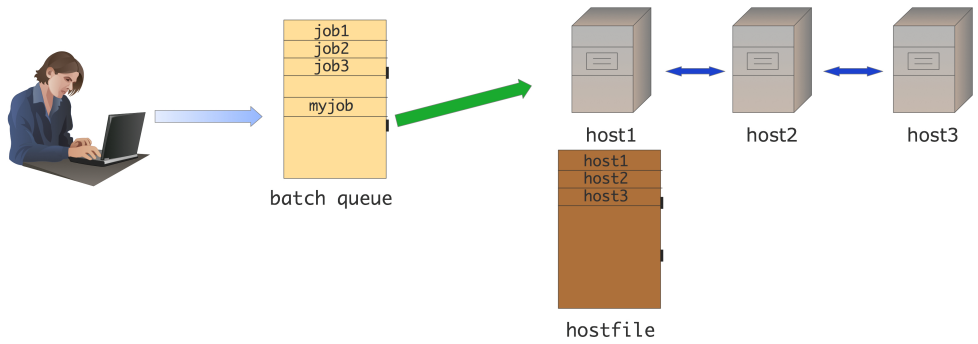


Figure 7.2 Batch MPI setup

hostfile is dynamically generated when the job starts. Since the job now runs at a time when the user may not be logged in, any screen output goes into an output file.

You see that in both scenarios the parallel program is started by the `mpirun` command using an Single Program Multiple Data (SPMD) mode of execution: all hosts execute the same program. It is possible for different hosts to execute different programs, but we will not consider that in this book.

There can be options and environment variables that are specific to some MPI installations, or to the network.

- *mpich* and its derivatives such as *Intel MPI* or *Cray MPI* have *mpiexec options*: `https://www.mpich.org/static/docs/v3.1/www1/mpiexec.html`

7.1.4 Making and running an MPI program

MPI is a library, called from programs in ordinary programming languages such as C/C++ or Fortran. To compile such a program you use your regular compiler:

```
gcc -c my_mpi_prog.c -I/path/to/mpi.h
gcc -o my_mpi_prog my_mpi_prog.o -L/path/to/mpi -lmpich
```

However, MPI libraries may have different names between different architectures, making it hard to have a portable makefile. Therefore, MPI typically has shell scripts around your compiler call:

```
mpicc -c my_mpi_prog.c
mpicc -o my_mpi_prog my_mpi_prog.o
```

MPI programs can be run on many different architectures. Obviously it is your ambition (or at least your dream) to run your code on a cluster with a hundred thousand processors and a fast network. But maybe you only have a small cluster with plain *ethernet*. Or maybe you're sitting in a plane, with just your laptop. An MPI program can be run in all these circumstances – within the limits of your available memory of course.

The way this works is that you do not start your executable directly, but you use a program, typically called `mpirun` or something similar, which makes a connection to all available processors and starts a run of your executable there. So if you have a thousand nodes in your cluster, `mpirun` can start your program once on each, and if you only have your laptop it can start a few instances there. In the latter case you will of course not get great performance, but at least you can test your code for correctness.

7.1.5 Language bindings

C/C++

The MPI library is written in C. Thus, its bindings are the most natural for that language.

C++ bindings existed at one point, but they were declared deprecated, and have been officially removed in the *MPI 3*. The *boost* library has its own version of MPI, but it seems not to be under further development. A recent effort at idiomatic C++ support is *MPL*[13].

Python

The *mpi4py* package [11] of *python bindings* is not defined by the MPI standards committee. Instead, it is the work of an individual, *Lisandro Dalcin*.

Notable about the Python bindings is that many communication routines exist in two variants:

- a version that can send native Python objects. These routines have lowercase names such as `bcast`; and
- a version that sends *numpy* objects; these routines have names such as `Bcast`. Their syntax can be slightly different.

The first version looks more 'pythonic', is easier to write, and can do things like sending python objects, but it is also decidedly less efficient since data is packed and unpacked with `pickle`. As a common sense guideline, use the `numpy` interface in the performance-critical parts of your code, and the native interface only for complicated actions in a setup phase.

Codes with `mpi4py` can be interfaced to other languages through Swig or conversion routines.

Data in `numpy` can be specified as a simple object, or `[data, (count,displ), datatype]`.

[13] See `http://numbercrunch.de/blog/2015/08/mpl-a-message-passing-library/`

7.2 Functional parallelism

MPI programs conform (mostly) to the Single Program Multiple Data (SPMD) model, where each processor runs the same executable. This running executable we call a *process*.

When MPI was first written, 20 years ago, it was clear what a processor was: it was what was in a computer on someone's desk, or in a rack. If this computer was part of a networked cluster, you called it a *node*. So if you ran an MPI program, each node would have one MPI process; figure 7.3. You could

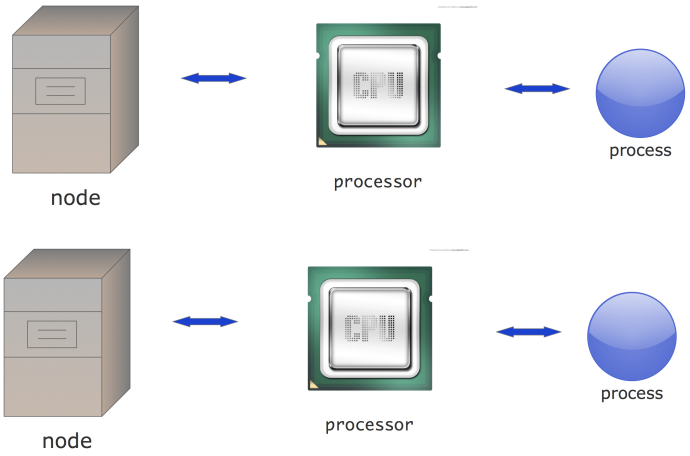


Figure 7.3 Cluster structure as of the mid 1990s

of course run more than one process, using the *time slicing* of the Operating System (OS), but that would give you no extra performance.

These days the situation is more complicated. You can still talk about a node in a cluster, but now a node can contain more than one processor chip (sometimes called a *socket*), and each processor chip probably has multiple *cores*. Figure 7.4 shows how you could explore this using a mix of MPI between the nodes, and a shared memory programming system on the nodes.

However, since each core can act like an independent processor, you can also have multiple MPI processes per node. To MPI, the cores look like the old completely separate processors. This is the 'pure MPI' model of figure 7.5, which we will use in most of this part of the book.

This is somewhat confusing: the old processors needed MPI programming, because they were physically separated. The cores on a modern processor, on the other hand, share the same memory, and even some caches. In its basic mode MPI seems to ignore all of this: each core receives an MPI process and the programmer writes the same send/receive call no matter where the other process is located. In fact, you can't immediately see whether two cores are on the same node or different nodes. Of course, on the implementation level MPI uses a different communication mechanism depending on whether cores are on the same socket or on different nodes, so you don't have to worry about lack of efficiency.

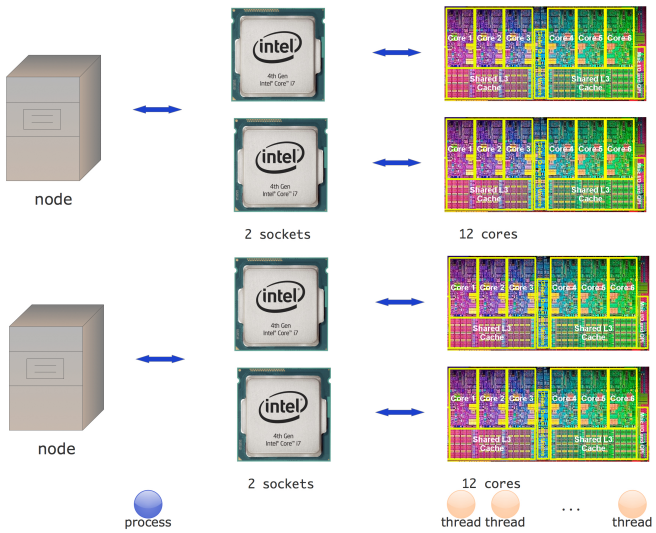


Figure 7.4 Hybrid cluster structure

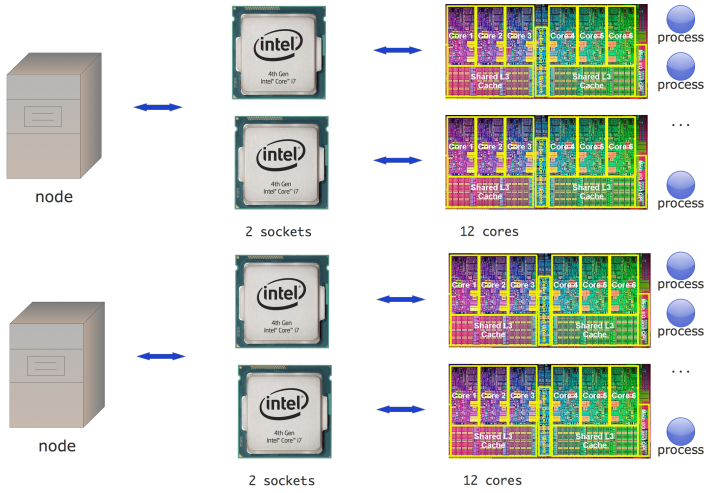


Figure 7.5 MPI-only cluster structure

In some rare cases you may want to run in an *Multiple Program Multiple Data (MPMD)* mode, rather than *SPMD*. This can be achieved either on the OS level, using options of the *mpiexec* mechanism, or you can use MPI's built-in process management.

7.2.1 Starting and running MPI processes

The SPMD model may be initially confusing. Even though there is only a single source, compiled into a single executable, the parallel run comprises a number of independently started MPI processes (see section 7.1.3 for the mechanism).

The following exercises are designed to give you an intuition for this one-source-many-processes setup. In the first exercise you will see that the mechanism for starting MPI programs starts up independent copies. There is nothing in the source that says 'and now you become parallel'.

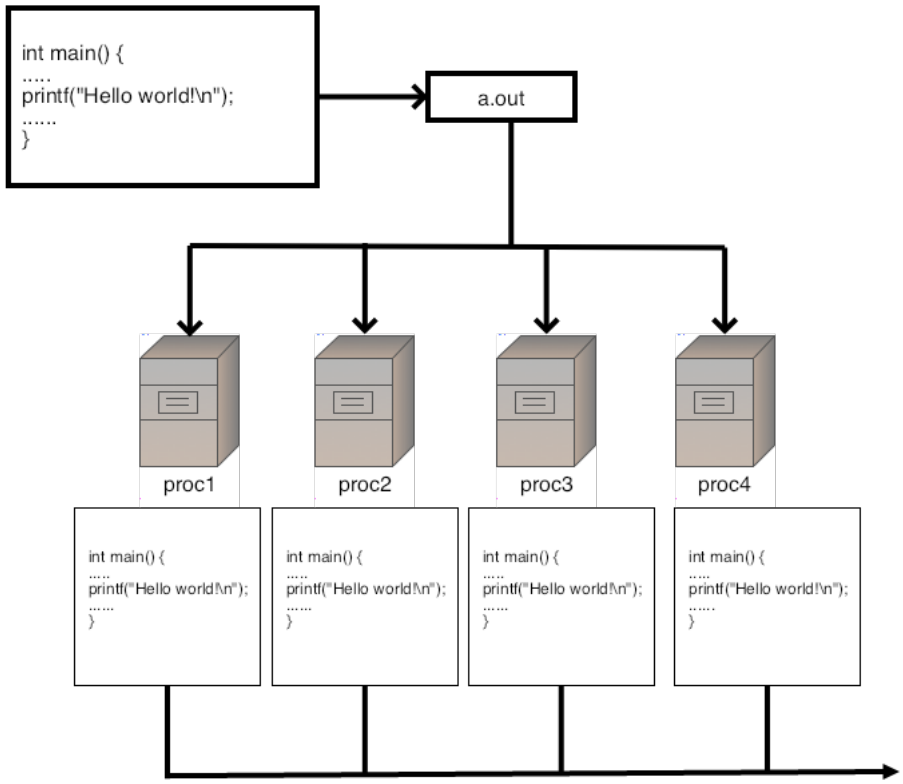


Figure 7.6 Running a hello world program in parallel

The following exercise demonstrates this point.

> **Exercise 7.1** Write a 'hello world' program, without any MPI in it, and run it in parallel with `mpiexec` or your local equivalent. Explain the output.

This exercise is illustrated in figure 7.6.

Headers

If you use MPI commands in a program file, be sure to include the proper header file, *mpi.h*.

```
#include "mpi.h" // for C
```

Initialization / finalization

To get a useful MPI program you need at least the calls `MPI_Init` and `MPI_Finalize` surrounding your code[14].

Every MPI program has to start with *MPI initialization* through `MPI_Init`, passing `argc` and `argv`, the arguments of a C language main program:

```
int main(int argc,char **argv) {
    ....
    return 0;
```

[14] In Python, there are no initialize and finalize calls: the `import` statement performs the initialization.

```
}
```

(It is allowed to pass `NULL` for these arguments.)

This may look a bit like declaring 'this is the parallel part of a program', but that's not true: again, the whole code is executed multiple times in parallel.

The regular way to conclude an MPI program is through `MPI_Finalize`.

> **Exercise 7.2** Add the commands `MPI_Init` and `MPI_Finalize` to your code. Put three different print statements in your code: one before the init, one between init and finalize, and one after the finalize. Again explain the output.

Aborting an MPI run: Apart from `MPI_Finalize`, which signals a successful conclusion of the MPI run, an abnormal end to a run can be forced by `MPI_Abort`. This aborts execution on all processes associated with the communicator, but many implementations simply abort all processes. The `value` parameter is returned to the environment.

Testing the initialized/finalized status: The commandline arguments `argc` and `argv` are only guaranteed to be passed to process zero, so the best way to pass commandline information is by a broadcast (section 7.3.5).

There are a few commands, such as `MPI_Get_processor_name`, that are allowed before `MPI_Init`.

If MPI is used in a library, MPI can have already been initialized in a main program. For this reason, one can test where `MPI_Init` has been called with `MPI_Initialized`.

You can test whether `MPI_Finalize` has been called with `MPI_Finalized`.

Commandline arguments: The `MPI_Init` routines takes a reference to `argc` and `argv` for the following reason: the `MPI_Init` calls filters out the arguments to *mpirun* or *mpiexec*, thereby lowering the value of `argc` and elimitating some of the `argv` arguments.

On the other hand, the commandline arguments that are meant for `mpiexec` wind up in the `MPI_INFO_ENV` object as a set of key/value pairs.

7.2.2 Processor identification

Since all processes in an MPI job are instantiations of the same executable, you'd think that they all execute the exact same instructions, which would not be terribly useful. You will now learn how to distinguish processes from each other, so that together they can start doing useful work.

Processor name

In the following exercise you will print out the hostname of each MPI process with `MPI_Get_processor_name` as a first way of distinguishing between processes.

> **Exercise 7.3** Now use the command `MPI_Get_processor_name` in between the init and finalize statement, and print out on what processor your process runs. Confirm that you are able to run a program that uses two different

nodes.

The character buffer needs to be allocated by you, it is not created by MPI, with size at least `MPI_MAX_PROCESSOR_NAME`.

The character storage is provided by the user: the character array must be at least `MPI_MAX_PROCESSOR_NAME` characters long. The actual length of the name is returned in the `resultlen` parameter.

Process and communicator properties: rank and size

First we need to introduce the concept of `communicator`, which is an abstract description of a group of processes. For now you only need to know about the existence of the `MPI_Comm` data type, and that there is a pre-defined communicator `MPI_COMM_WORLD` which describes all the processes involved in your parallel run.

To distinguish between processes in a communicator, MPI provides two calls

1. `MPI_Comm_size` reports how many processes there are in all; and
2. `MPI_Comm_rank` states what the number of the process is that calls this routine.

If every process executes the `MPI_Comm_size` call, they all get the same result, namely the total number of processes in your run. On the other hand, if every process executes `MPI_Comm_rank`, they all get a different result, namely their own unique number, an integer in the range from zero to the the number of processes minus 1. See figure 7.7. In other words, each process can find out

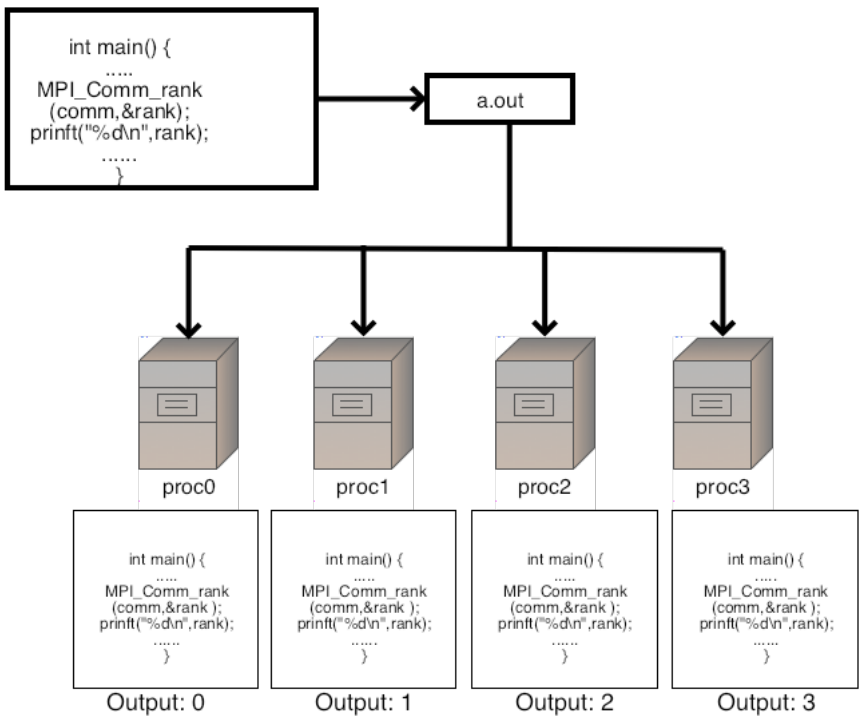


Figure 7.7 Parallel program that prints process rank

'I am process 5 out of a total of 20'.

Exercise 7.4 Write a program where each process prints out a message reporting its number, and how many processes there are:

```
Hello from process 2 out of 5!
```

Write a second version of this program, where each process opens a unique file and writes to it. *On some clusters this may not be advisable if you have large numbers of processors, since it can overload the file system.*

Exercise 7.5 Write a program where only the process with number zero reports on how many processes there are in total.

7.2.3 Functional parallelism

Being able to tell processes apart is already enough for some applications. Based on its rank, a processor can find its section of a search space. For instance, in *Monte Carlo codes* a large number of random samples is generated and some computation performed on each. (This particular example requires each MPI process to run an independent random number generator, which is not entirely trivial.)

As another example, in *Boolean satisfiability* problems a number of points in a search space needs to be evaluated. Knowing a process's rank is enough to let it generate its own portion of the search space. The computation of the *Mandelbrot set* can also be considered as a case of functional parallelism. However, the image that is constructed is data that needs to be kept on one processor, which breaks the symmetry of the parallel run.

Of course, at the end of a functionally parallel run you need to summarize the results, for instance printing out some total. The mechanisms for that you will learn next.

7.3 Collectives

If all processes have individual data, for instance the result of a local computation, you may want to bring that information together, for instance to find the maximal computed value or the sum of all values. Conversely, sometimes one processor has information that needs to be shared with all. For this sort of operation, MPI has `collectives`.

There are various cases, illustrated in figure 7.8, which you can (sort of) motivated by considering some classroom activities:

- The teacher tells the class when the exam will be. This is a `broadcast`: the same item of information goes to everyone.
- After the exam, the teacher performs a `gather` operation to collect the invidivual exams.
- On the other hand, when the teacher computes the average grade, each student has an individual number, but these are now combined to compute

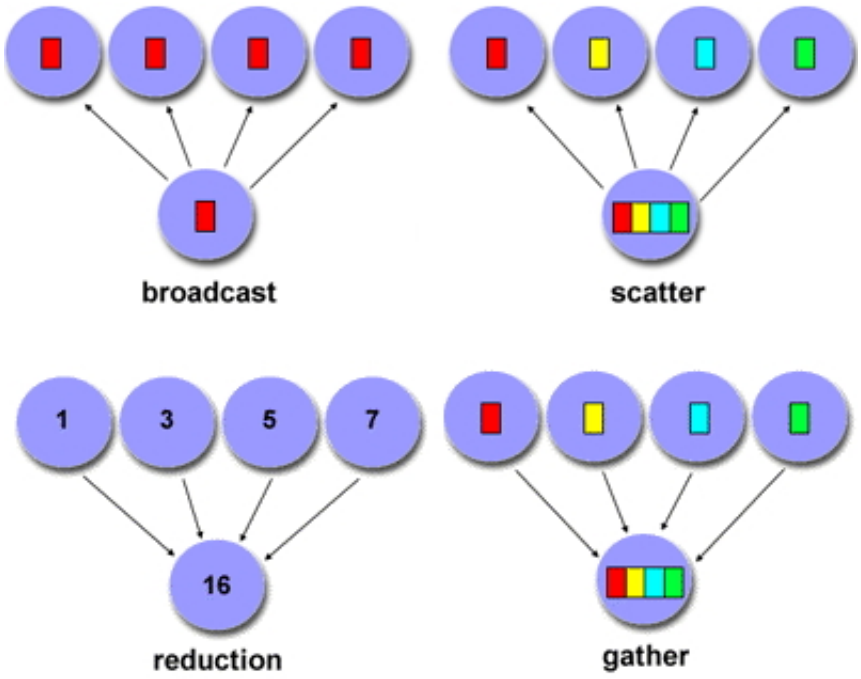


Figure 7.8 The four most common collectives

a single number. This is a `reduction`.

- Now the teacher has a list of grades and gives each student their grade. This is a `scatter` operation, where one process has multiple data items, and gives a different one to all the other processes.

This story is a little different from what happens with MPI processes, because these are more symmetric; the process doing the reducing and broadcasting is no different from the others. Any process can function as the *root process* in such a collective.

Exercise 7.6 How would you realize the following scenarios with MPI collectives?

- Let each process compute a random number. You want to print the maximum of these numbers to your screen.
- Each process computes a random number again. Now you want to scale these numbers by their maximum.
- Let each process compute a random number. You want to print on what processor the maximum value is computed.

7.3.1 Practical use of collectives

Collectives are quite common in scientific applications. For instance, if one process reads data from disc or the commandline, it can use a broadcast or a gather to get the information to other processes. Likewise, at the end of a program run, a gather or reduction can be used to collect summary information about the program run.

However, a more common scenario is that the result of a collective is needed on all processes.

Consider the computation of the *standard deviation*:

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i}^{N}(x_i-\mu)^2} \qquad \text{where} \qquad \mu = \frac{\sum_i^N x_i}{N}$$

and assume that every process stores just one x_i value.

1. The calculation of the average μ is a reduction, since all the distributed values need to be added.
2. Now every process needs to compute $x_i - \mu$ for its value x_i, so μ is needed everywhere. You can compute this by doing a reduction followed by a broadcast, but it is better to use a so-called *allreduce* operation, which does the reduction and leaves the result on all processors.
3. The calculation of $\sum_i(x_i-\mu)$ is another sum of distributed data, so we need another reduction operation. Depending on whether each process needs to know σ, we can again use a allreduce.

7.3.2 Synchronization

Collectives are operations that involve all processes in a communicator. A collective is a single call, and it blocks on all processors. That does not mean that all processors exit the call at the same time: because of implementational details and network latency they need not be synchronized in their execution. However, semantically we can say that a process can not finish a collective until every other process has at least started the collective.

In addition to these collective operations, there are operations that are said to be 'collective on their communicator', but which do not involve data movement. Collective then means that all processors must call this routine; not to do so is an error that will manifest itself in 'hanging' code.

7.3.3 Collectives in MPI

We will now explain the MPI collectives in the following order.

Allreduce We use the allreduce as an introduction to the concepts behind collectives; section 7.3.4. As explained above, this routines servers many practical scenarios.

Broadcast and reduce We then introduce the concept of a root in the reduce (section 7.3.5) and broadcast (section 7.3.5) collectives.

Gather and scatter The gather/scatter collectives deal with more than a single data item.

There are more collectives or variants on the above.

- If you want to gather or scatter information, but the contribution of each processor is of a different size, there are 'variable' collectives; they have a `v` in the name.

- Sometimes you want a reduction with partial results, where each processor computes the sum (or other operation) on the values of lower-numbered processors. For this, you use a *scan* collective.
- If every processor needs to broadcast to every other, you use an *all-to-all* operation.
- A barrier is an operation that makes all processes wait until every process has reached the barrier.

7.3.4 Reduction

Reduce to all

Above we saw a couple of scenarios where a quantity is reduced, with all proceses getting the result. The MPI call for this is `MPI_Allreduce`.

Example: we give each process a random number, and sum these numbers together. The result should be approximately 1/2 times the number of processes.

```
// allreduce.c
float myrandom,sumrandom;
myrandom = (float) rand()/(float)RAND_MAX;
// add the random variables together
MPI_Allreduce(&myrandom,&sumrandom,
                1,MPI_FLOAT,MPI_SUM,comm);
// the result should be approx nprocs/2:
if (procno==nprocs-1)
  printf("Result %6.9f compared to .5\n",sumrandom/nprocs);
```

For Python we illustrate both the native and the numpy variant. In the numpy variant we create an array for the receive buffer, even though only one element is used.

```
## allreduce.py
random_number = random.randint(1,nprocs*nprocs)
print("[%d] random=%d" % (procid,random_number))

max_random = comm.allreduce(random_number,op=MPI.MAX)
if procid==0:
    print("Python native:\n  max=%d" % max_random)

myrandom = np.empty(1,dtype=np.int)
myrandom[0] = random_number
allrandom = np.empty(nprocs,dtype=np.int)

comm.Allreduce(myrandom,allrandom[:1],op=MPI.MAX)
```

Exercise 7.7 Let each process compute a random number, and compute the sum of these numbers using the `MPI_Allreduce` routine.

(The operator is `MPI_SUM` for C, or `MPI.SUM` for Python.)

Each process then scales its value by this sum. Compute the sum of the scaled numbers and check that it is 1.

Reduction of distributed data

One of the more common applications of the reduction operation is the *inner product* computation. Typically, you have two vectors x, y that have the same distribution, that is, where all processes store equal parts of x and y. The computation is then

```
local_inprod = 0;
for (i=0; i<localsize; i++)
  local_inprod += x[i]*y[i];
MPI_Allreduce( &local_inprod, &global_inprod, 1,MPI_DOUBLE ... )
```

Reduce in place

By default MPI will not overwrite the original data with the reduction result, but you can tell it to do so using the `MPI_IN_PLACE` specifier:

```
// allreduceinplace.c
int nrandoms = 500000;
float *myrandoms;
myrandoms = (float*) malloc(nrandoms*sizeof(float));
for (int irand=0; irand<nrandoms; irand++)
  myrandoms[irand] = (float) rand()/(float)RAND_MAX;
// add all the random variables together
MPI_Allreduce(MPI_IN_PLACE,myrandoms,
              nrandoms,MPI_FLOAT,MPI_SUM,comm);
// the result should be approx nprocs/2:
if (procno==nprocs-1) {
  float sum=0.;
  for (int i=0; i<nrandoms; i++) sum += myrandoms[i];
  sum /= nrandoms*nprocs;
  printf("Result %6.9f compared to .5\n",sum);
}
```

This has the advantage of saving half the memory.

7.3.5 Rooted collectives: broadcast, reduce

In some scenarios there is a certain process that has a privileged status.

- One process can generate or read in the initial data for a program run. This then needs to be communicated to all other processes.
- At the end of a program run, often one process needs to output some summary information.

This process is called the *root* process, and we will now consider routines that have a root.

Reduce to a root

In the broadcast operation a single data item was communicated to all processes. A reduction operation with `MPI_Reduce` goes the other way: each process has a data item, and these are all brought together into a single item.

Here are the essential elements of a reduction operation:

```
MPI_Reduce( senddata, recvdata..., operator, root, comm );
```

- There is the original data, and the data resulting from the reduction. It is a design decision of MPI that it will not by default overwrite the original data. The send data and receive data are of the same size and type: if every processor has one real number, the reduced result is again one real number.
- There is a reduction operator. Popular choices are `MPI_SUM`, `MPI_PROD` and `MPI_MAX`, but complicated operators such as finding the location of the maximum value exist. You can also define your own operators.
- There is a root process that receives the result of the reduction. Since the non-root processes do not receive the reduced data, they can actually leave the receive buffer undefined.

```
// reduce.c
float myrandom = (float) rand()/(float)RAND_MAX,
  result;
int target_proc = nprocs-1;
// add all the random variables together
MPI_Reduce(&myrandom,&result,1,MPI_FLOAT,MPI_SUM,
           target_proc,comm);
// the result should be approx nprocs/2:
if (procno==target_proc)
  printf("Result %6.3f compared to nprocs/2=%5.2f\n",
         result,nprocs/2.);
```

Collective operations can also take an array argument, instead of just a scalar. In that case, the operation is applied pointwise to each location in the array.

Reduce in place

Instead of using a send and a receive buffer in the reduction, it is possible to avoid the send buffer by putting the send data in the receive buffer. We see this mechanism in section 7.3.4 for the allreduce operation.

For the rooted call `MPI_Reduce`, it is similarly possible to use the value in the receive buffer on the root. However, on all other processes, data is placed in the send buffer and the receive buffer is null or ignored as before.

This example sets the buffer values through some pointer cleverness in order to have the same reduce call on all processes.

```
// reduceinplace.c
float mynumber,result,*sendbuf,*recvbuf;
mynumber = (float) procno;
```

```
int target_proc = nprocs-1;
// add all the random variables together
if (procno==target_proc) {
  sendbuf = (float*)MPI_IN_PLACE; recvbuf = &result;
  result = mynumber;
} else {
  sendbuf = &mynumber;    recvbuf = NULL;
}
MPI_Reduce(sendbuf,recvbuf,1,MPI_FLOAT,MPI_SUM,
           target_proc,comm);
// the result should be nprocs*(nprocs-1)/2:
if (procno==target_proc)
  printf("Result %6.3f compared to n(n-1)/2=%5.2f\n",
         result,nprocs*(nprocs-1)/2.);
```

Broadcast

The broadcast call has the following structure:

```
MPI_Bcast( data..., root , comm);
```

The root is the process that is sending its data. Typically, it will be the root of a broadcast tree. The `comm` argument is a communicator: for now you can use `MPI_COMM_WORLD`. Unlike with send/receive there is no message tag, because collectives are blocking, so you can have only one collective active at a time.

The data in a broadcast with `MPI_Bcast` (or any other MPI operation for that matter) is specified as

- A buffer. In C this is the address in memory of the data. This means that you broadcast a single scalar as `MPI_Bcast( &value, ... )`, but an array as `MPI_Bcast( array, ... )`.
- The number of items and their datatype. The allowable datatypes are such things as `MPI_INT` and `MPI_FLOAT` for C, or more complicated types.

Example: in general we can not assume that all processes get the commandline arguments, so we broadcast them from process 0.

```
// init.c
if (procno==0) {
  if ( argc==1 || // the program is called without parameter
       ( argc>1 && !strcmp(argv[1],"-h") ) // user asked for help
       ) {
    printf("\nUsage: init [0-9]+\n");
    MPI_Abort(comm,1);
  }
  input_argument = atoi(argv[1]);
}
MPI_Bcast(&input_argument,1,MPI_INT,0,comm);
```

Exercise 7.8 If you give a commandline argument to a program, that argument is available as a character string as part of the `argv,argc` pair that you typically use as the arguments to your main program. You can use the function `atoi` to convert such a string to integer.

Write a program where process 0 looks for an integer on the commandline, and broadcasts it to the other processes. Initialize the buffer on all processes, and let all processes print out the broadcast number, just to check that you solved the problem correctly.

In python we illustrate the native and numpy variants. In the native variant the result is given as a function return; in the numpy variant the send buffer is reused.

```
## bcast.py
# first native
if procid==root:
    buffer = [ 5.0 ] * dsize
else:
    buffer = [ 0.0 ] * dsize
buffer = comm.bcast(obj=buffer,root=root)
if not reduce( lambda x,y:x and y,
               [ buffer[i]==5.0 for i in  range(len(buffer)) ] ):
    print( "Something wrong on proc %d: native buffer <<%s>>" \
           % (procid,str(buffer)) )

# then with NumPy
buffer = np.arange(dsize, dtype=np.float64)
if procid==root:
    for i in range(dsize):
        buffer[i] = 5.0
comm.Bcast( buffer,root=root )
if not all( buffer==5.0 ):
    print( "Something wrong on proc %d: numpy buffer <<%s>>" \
           % (procid,str(buffer)) )
else:
    if procid==root:
        print("Success.")
```

7.3.6 Rooted collectives: gather and scatter

In the `MPI_Scatter` operation, the root spreads information to all other processes. The difference with a broadcast is that it involves individual information from/to every process. Thus, the gather operation typically has an array of items, one coming from each sending process, and scatter has an array, with an individual item for each receiving process; see figure 7.10.

These gather and scatter collectives have a different parameter list from the broadcast/reduce. The broadcast/reduce involves the same amount of data

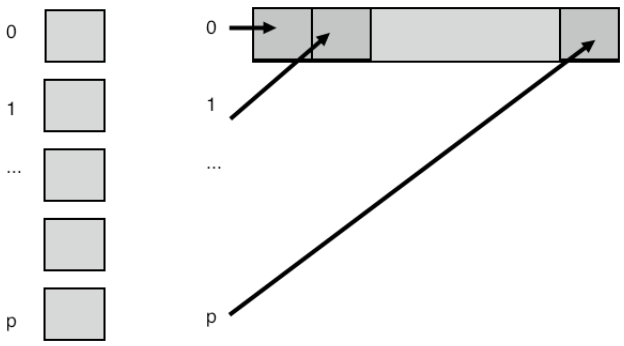


Figure 7.9 Gather collects all data onto a root

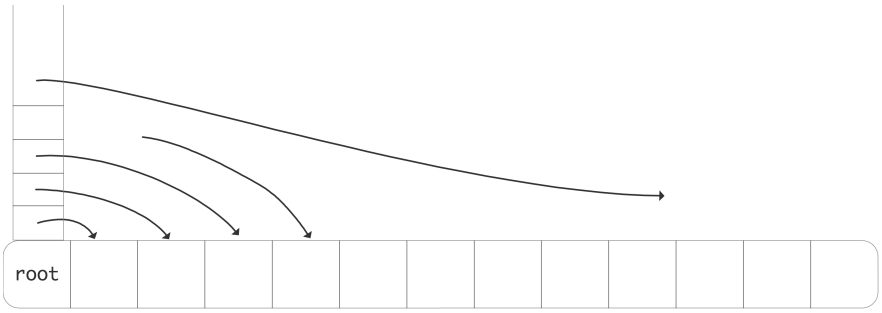


Figure 7.10 A scatter operation

on each process, so it was enough to have a buffer, datatype, and size. In the gather/scatter calls you have

- a large buffer on the root, with a datatype and size specification, and
- a smaller buffer on each process, with its own type and size specification.

Of course, since we're in SPMD mode, even non-root processes have the argument for the send buffer, but they ignore it. For instance:

```
int MPI_Scatter
  (void* sendbuf, int sendcount, MPI_Datatype sendtype,
  void* recvbuf, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm)
```

The `sendcount` is not, as you might expect, the total length of the sendbuffer; instead, it is the amount of data sent to each process.

In the gather and scatter calls, each processor has n elements of individual data. There is also a root processor that has an array of length np, where p is the number of processors. The gather call collects all this data from the processors to the root; the scatter call assumes that the information is initially on the root and it is spread to the individual processors.

The prototype for `MPI_Gather` has two 'count' parameters, one for the length of the individual send buffers, and one for the receive buffer. However, confusingly, the second parameter (which is only relevant on the root) does not indicate the total amount of information coming in, but rather the size of *each* contribution. Thus, the two count parameters will usually be the same (at least on the root); they can differ if you use different `MPI_Datatype` values for

the sending and receiving processors.

Here is a small example:

```
// gather.c
// we assume that each process has a value "localsize"
// the root process collectes these values

if (procno==root)
  localsizes = (int*) malloc( (nprocs+1)*sizeof(int) );

// everyone contributes their info
MPI_Gather(&localsize,1,MPI_INT,
           localsizes,1,MPI_INT,root,comm);
```

The `MPI_IN_PLACE` option can be used for the send buffer on the root; the data for the root is then assumed to be already in the correct location in the receive buffer.

The `MPI_Scatter` operation is in some sense the inverse of the gather: the root process has an array of length np where p is the number of processors and n the number of elements each processor will receive.

```
int MPI_Scatter
  (void* sendbuf, int sendcount, MPI_Datatype sendtype,
  void* recvbuf, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm)
```

Example

In some applications, each process computes a row or column of a matrix, but for some calculation (such as the determinant) it is more efficient to have the whole matrix on one process. You should of course only do this if this matrix is essentially smaller than the full problem, such as an interface system or the last coarsening level in multigrid.

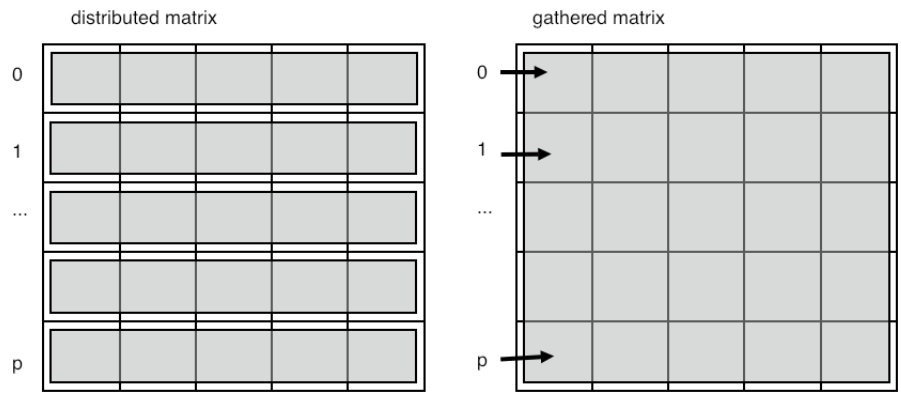


Figure 7.11 Gather a distributed matrix onto one process

Figure 7.11 pictures this. Note that conceptually we are gathering a two-dimensional object, but the buffer is of course one-dimensional. You will later

see how this can be done more elegantly with the 'subarray' datatype.

Allgather

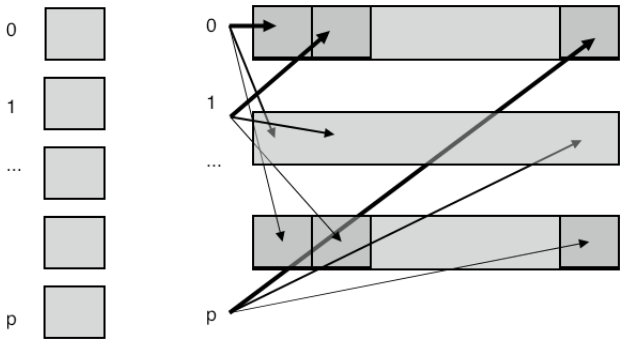


Figure 7.12 All gather collects all data onto every process

The `MPI_Allgather` routine does the same gather onto every process: each process winds up with the totality of all data; figure 7.12.

This routine can be used in the simplest implementation of the *dense matrix-vector product* to give each processor the full input.

Some cases look like an all-gather but can be implemented more efficiently. Suppose you have two distributed vectors, and you want to create a new vector that contains those elements of the one that do not appear in the other. You could implement this by gathering the second vector on each processor, but this may be prohibitive in memory usage.

7.3.7 MPI Operators

MPI *operators* are used in reduction operators. Most common operators, such as sum or maximum, have been built into the MPI library, but it is possible to define new operators.

Pre-defined operators

The following is the list of *pre-defined operators* `MPI_OP` values.

MPI type	meaning	applies to
`MPI_MAX`	maximum	integer, floating point
`MPI_MIN`	minimum	
`MPI_SUM`	sum	integer, floating point, complex, multilanguage types
`MPI_PROD`	product	
`MPI_LAND`	logical and	C integer, logical
`MPI_LOR`	logical or	
`MPI_LXOR`	logical xor	
`MPI_BAND`	bitwise and	integer, byte, multilanguage types
`MPI_BOR`	bitwise or	
`MPI_BXOR`	bitwise xor	
`MPI_MAXLOC`	max value and location	`MPI_DOUBLE_INT` and such
`MPI_MINLOC`	min value and location	

The `MPI_MAXLOC` operation yields both the maximum and the rank on which it occurs. However, to use it the input should be an array of `real/int` structs, where the `int` is the rank of the number.

7.4 Distributed computing and distributed data

One reason for using MPI is that sometimes you need to work on more data than can fit in the memory of a single processor. With distributed memory, each processor then gets a part of the whole data structure and only works on that.

So let's say we have a large array, and we want to distribute the data over the processors. That means that, with `p` processes and `n` elements per processor, we have a total of $n \cdot p$ elements.

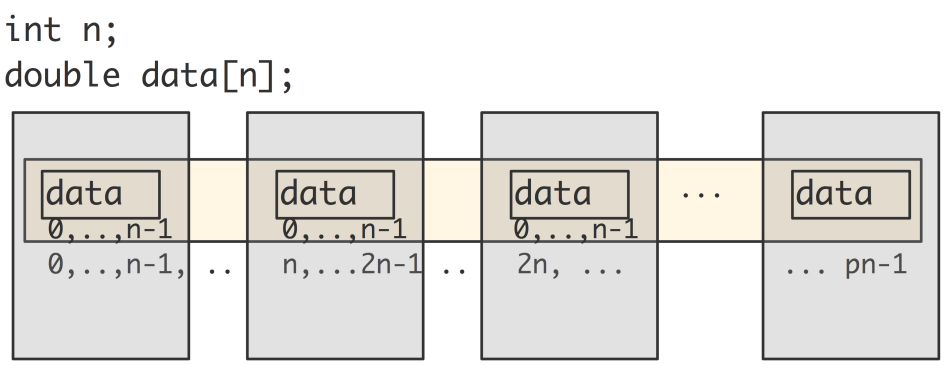


Figure 7.13 Local parts of a distributed array

We sometimes say that `data` is the local part of a *distributed array* with a total size of $n \cdot p$ elements. However, this array only exists conceptually: each processor has an array with lowest index zero, and you have to translate that yourself to an index in the global array. In other words, you have to write your code in such a way that it acts like you're working with a large array that is distributed over the processors, while actually manipulating only the local arrays on the processors.

Your typical code then looks like

```
int myfirst = .....;
for (int ilocal=0; ilocal<nlocal; ilocal++) {
  int iglobal = myfirst+ilocal;
  array[ilocal] = f(iglobal);
}
```

Exercise 7.9 We want to compute $\sum_{n=1}^{N} n^2$, and we do that as follows by filling in an array and summing the elements. (Yes, you can do it without an array, but for purposes of the exercise do it with.)

Set a variable `N` for the total length of the array, and compute the local number of elements. Make sure you handle the case where N does not divide

perfectly by the number of processes.

- Now allocate the local parts: each processor should allocate only local elements, not the whole vector.
  (Allocate your array as real numbers. Why are integers not a good idea?)
- On each processor, initialize the local array so that the i-th location of the distributed array (for $i = 0, \ldots, N-1$) contains $(i+1)^2$.
- Now use a collective operation to compute the sum of the array values. The right value is $(2N^3 + 3N^2 + N)/6$. Is that what you get?

(Note that computer arithmetic is not exact: the computed sum will only be accurate up to some relative accuracy.)

If the array size is not perfectly divisible by the number of processors, we have to come up with a division that is uneven, but not too much. You could for instance, write

```
int Nglobal, // is something large
    Nlocal = Nglobal/ntids,
    excess = Nglobal%ntids;
if (mytid==ntids-1)
  Nlocal += excess;
```

Exercise 7.10 Argue that this strategy is not optimal. Can you come up with a better distribution? Load balancing is further discussed in section 4.7.

Exercise 7.11 Implement an inner product routine: let x be a distributed vector of size N with elements $x[i] = i$, and compute $x^t x$. As before, the right value is $(2N^3 + 3N^2 + N)/6$.

Use the inner product value to scale to vector so that it has norm 1. Check that your computation is correct.

7.4.1 Blocking point-to-point operations

Suppose you have an array of numbers $x_i : i = 0, \ldots, N$ and you want to compute $y_i = (x_{i-1} + x_i + x_{i+1})/3 : i = 1, \ldots, N-1$. As before (see figure 7.13), we give each processor a subset of the x_is and y_is. Let's define i_p as the first index of y that is computed by processor p. (What is the last index computed by processor p? How many indices are computed on that processor?)

We often talk about the *owner computes* model of parallel computing: each processor 'owns' certain data items, and it computes their value.

Now let's investigate how processor p goes about computing y_i for the i-values it owns. Let's assume that processor p also stores the values x_i for these same indices. Now, for many values it can compute

$$y_i = (x_{i-1} + x_i + x_{i+1})/3$$

(figure 7.15). However, there is a problem with computing the first index i_p:

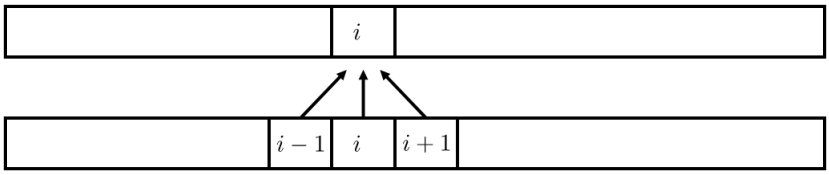


Figure 7.14 Three point averaging

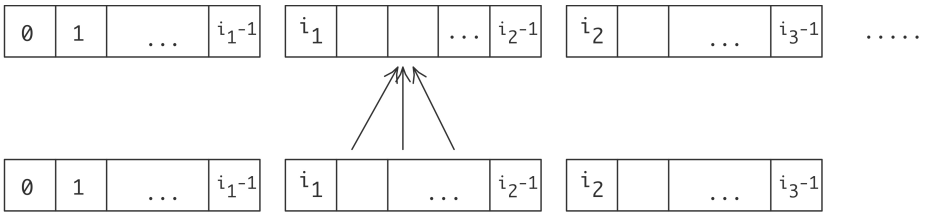


Figure 7.15 Three point averaging in parallel

$$y_{i_p} = (x_{i_p-1} + x_{i_p} + x_{i_p+1})/3$$

since x_{i_p} is not stored on processor p: it is stored on $p-1$ (figure 7.16). There

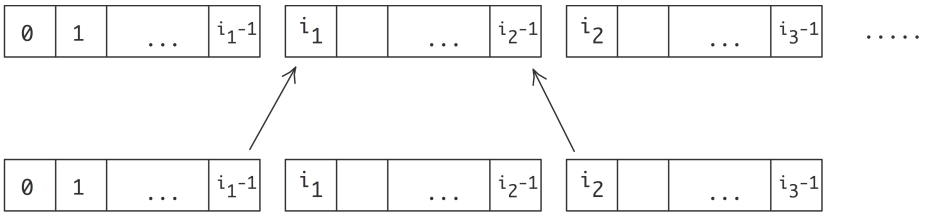


Figure 7.16 Three point averaging in parallel, case of edge points

is a similar story with the last index that p tries to compute: that involves a value that is only present on $p+1$.

You see that there is a need for processor-to-processor, or technically *point-to-point*, information exchange. MPI realizes this through matched send and receive calls:

- One process does a send to a specific other process;
- the other process does a specific receive from that source.

Send example: ping-pong

A simple scenario for information exchange between just two processes is the *ping-pong*: process A sends data to process B, which sends data back to A. This means that process A executes the code

```
MPI_Send( /* to: */ B ..... );
MPI_Recv( /* from: */ B ... );
```

while process B executes

```
MPI_Recv( /* from: */ A ... );
MPI_Send( /* to: */ A ..... );
```

Since we are programming in SPMD mode, this means our program looks like:

```
if ( /* I am process A */ ) {
  MPI_Send( /* to: */ B ..... );
  MPI_Recv( /* from: */ B ... );
} else if ( /* I am process B */ ) {
  MPI_Recv( /* from: */ A ... );
  MPI_Send( /* to: */ A ..... );
}
```

Send call The blocking send command is `MPI_Send`. The features of this call are:

- *Data description* The data to be sent is described by a trio of buffer/count/-datatype. This is common to just about any MPI routine that involves data transfer.

In C the buffer is a memory address, so it is treated slightly differently between variables and arrays:

```
int counter;
MPI_Send( &counter,1,MPI_INT, /* ... */ );
float point[2];
MPI_Send( point,2,MPI_FLOAT, /* ... */ );
```

Python sends objects, which document their own size and type, so the send call has only a single parameter describing the data:

```
comm.send( python_object, ... )
comm.Send( numpy_object, ... )
```

- *Target* Send calls have a *messsage target*: they require an explicit process rank to send to. This rank is a number from zero up to the result of `MPI_Comm_size`.

This aspect of the send call shows the symmetric nature of MPI: every target process is reached with the same send call, no matter whether it's running on the same multicore chip as the sender, or on a computational node halfway across the machine room. Of course, any self-respecting MPI implementation optimizes for the case where sender and receiver have access to the same shared memory. However, even then, there will be a copy operation from the sender buffer to the receiver buffer, so there is no actual memory sharing going on.

- *Tag* Many applications have each sender send only one message to a given receiver. For the case where there are multiple messages between the same sender / receiver pair, the *message tag* can be used to disambiguate between the messages.

Unless otherwise needed, a tag value of zero is safe to use. If you do use tag values, you can use the key `MPI_TAG_UB` to query what the maximum value is that can be used.

Receive call The basic blocking receive command is `MPI_Recv`

- *Data description* The receive call has the same buffer/count/data parameters as the send call. However, the `count` argument here indicates the maximum

length of a message; the actual length of the received message can be determined from the status object.

Source Mirroring the target argument of the `MPI_Send` call, `MPI_Recv` has a *messagesource* argument. This can be either a specific process rank, or it can be the `MPI_ANY_SOURCE` wildcard. In the latter case, the actual source can be determined after the message has been received.

Tag Similar to the messsage source, the message tag of a receive call can be a specific value or a wildcard, in this case `MPI_ANY_TAG`.

Status In the syntax of the `MPI_Recv` command you saw one parameter that the send call lacks: the `MPI_Status` object. This serves the following purpose: the receive call can have a 'wildcard' behaviour, for instance specifying that the message can come from any source rather than a specific one. Similarly, the tag value can be wildcarded. The status object then allows you to find out the actual message source and tag, and the message size.

Exercise 7.12 Implement the ping-pong program. Add a timer using `MPI_Wtime`. For the `status` argument of the receive call, use `MPI_STATUS_IGNORE`.

- Run multiple ping-pongs (say a thousand) and put the timer around the loop. The first run may take longer; try to discard it.
- Run your code with the two communicating processes first on the same node, then on different nodes. Do you see a difference?
- Then modify the program to use longer messages. How does the timing increase with message size?

For bonus points, can you do a regression to determine α, β?

Exercise 7.13 Take your pingpong program and modify it to let half the processors be source and the other half the targets. Does the pingpong time increase?

Problems with blocking communication

The use of `MPI_Send` and `MPI_Recv` is known as *blocking communication*: when your code reaches a send or receive call, it blocks until the call is succesfully completed. For a receive call it is clear that the receiving code will wait until the data has actually come in, but for a send call this is more subtle.

You may be tempted to think that the send call puts the data somewhere in the network, and the sending code can progress, as in figure 7.17, left. But

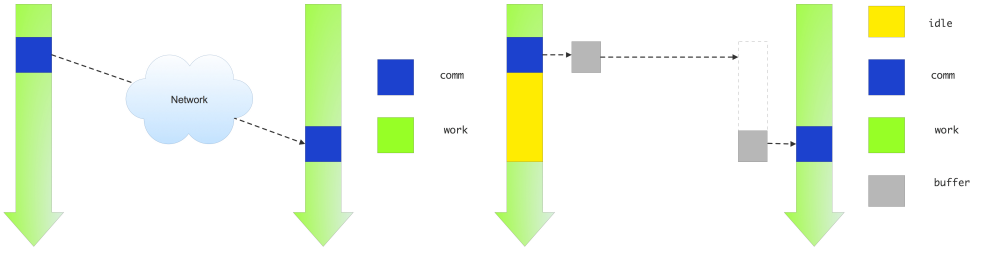


Figure 7.17 Illustration of an ideal (left) and actual (right) send-receive interaction

this ideal scenario is not realistic: it assumes that somewhere in the network

there is buffer capacity for all messages that are in transit. This is not the case: data resides on the sender, and the sending call blocks, until the receiver has received all of it. (There is a exception for small messages, as explained in the next section.)

Deadlock Suppose two process need to exchange data, and consider the following pseudo-code, which purports to exchange data between processes 0 and 1:

```
other = 1-mytid; /* if I am 0, other is 1; and vice versa */
receive(source=other);
send(target=other);
```

Imagine that the two processes execute this code. They both issue the send call... and then can't go on, because they are both waiting for the other to issue a receive call. This is known as *deadlock*.

Eager limit If you reverse the send and receive call, you should get deadlock, but in practice that code will often work. The reason is that MPI implementations sometimes send small messages regardless of whether the receive has been posted. This relies on the availability of some amount of available buffer space. The size under which this behaviour is used is sometimes referred to as the *eager limit*.)

The following code is guaranteed to block, since a `MPI_Recv` always blocks:

```
// recvblock.c
other = 1-procno;
MPI_Recv(&recvbuf,1,MPI_INT,other,0,comm,&status);
MPI_Send(&sendbuf,1,MPI_INT,other,0,comm);
printf("This statement will not be reached on %d\n",procno);
```

On the other hand, if we put the send call before the receive, code may not block for small messages that fall under the eager limit.

To illustrate eager and blocking behavior in `MPI_Send`, consider an example where we send gradually larger messages. From the screen output you can see what the largest message was that fell under the eager limit; after that the code hangs because of a deadlock.

```
// sendblock.c
other = 1-procno;
/* loop over increasingly large messages */
for (int size=1; size<2000000000; size*=10) {
  sendbuf = (int*) malloc(size*sizeof(int));
  recvbuf = (int*) malloc(size*sizeof(int));
  if (!sendbuf || !recvbuf) {
    printf("Out of memory\n"); MPI_Abort(comm,1);
  }
  MPI_Send(sendbuf,size,MPI_INT,other,0,comm);
  MPI_Recv(recvbuf,size,MPI_INT,other,0,comm,&status);
  /* If control reaches this point, the send call
```

```
      did not block. If the send call blocks,
      we do not reach this point, and the program will hang.
  */
  if (procno==0)
    printf("Send did not block for size %d\n",size);
  free(sendbuf); free(recvbuf);
}
```

In Python:

```
## sendblock.py
size = 1
while size<2000000000:
    sendbuf = np.empty(size, dtype=np.int)
    recvbuf = np.empty(size, dtype=np.int)
    comm.Send(sendbuf,dest=other)
    comm.Recv(sendbuf,source=other)
    if procid<other:
        print("Send did not block for",size)
    size *= 10
```

If you want a code to exhibit the same blocking behavior for all message sizes, you force the send call to be blocking by using `MPI_Ssend`, which has the same calling sequence as `MPI_Send`.

```
// ssendblock.c
other = 1-procno;
sendbuf = (int*) malloc(sizeof(int));
recvbuf = (int*) malloc(sizeof(int));
size = 1;
MPI_Ssend(sendbuf,size,MPI_INT,other,0,comm);
MPI_Recv(recvbuf,size,MPI_INT,other,0,comm,&status);
printf("This statement is not reached\n");
```

Formally you can describe deadlock as follows. Draw up a graph where every process is a node, and draw a directed arc from process A to B if A is waiting for B. There is deadlock if this directed graph has a loop.

The solution to the deadlock in the above example is to first do the send from 0 to 1, and then from 1 to 0 (or the other way around). So the code would look like:

```
if ( /* I am processor 0 */ ) {
  send(target=other);
  receive(source=other);
} else {
  receive(source=other);
  send(target=other);
}
```

The eager limit is implementation-specific. For instance, for *Intel mpi* there is a variable `I_MPI_EAGER_THRESHOLD`, for *mvapich2* it is `MV2_IBA_EAGER_THRESHOLD`, and for *OpenMPI* the `-mca` options `btl_openib_eager_limit` and `btl_openib_rndv_eager_limit`.

Serialization There is a second, even more subtle problem with blocking communication. Consider the scenario where every processor needs to pass data to its successor, that is, the processor with the next higher rank. The basic idea would be to first send to your successor, then receive from your predecessor. Since the last processor does not have a successor it skips the send, and likewise the first processor skips the receive. The pseudo-code looks like:

```
successor = mytid+1; predecessor = mytid-1;
if ( /* I am not the last processor */ )
  send(target=successor);
if ( /* I am not the first processor */ )
  receive(source=predecessor)
```

This code does not deadlock. All processors but the last one block on the send call, but the last processor executes the receive call. Thus, the processor before the last one can do its send, and subsequently continue to its receive, which enables another send, et cetera.

In one way this code does what you intended to do: it will terminate (instead of hanging forever on a deadlock) and exchange data the right way. However, the execution now suffers from unexpected *serialization*: only one processor is active at any time, so what should have been a parallel operation becomes a sequential one. This is illustrated in figure 7.18.

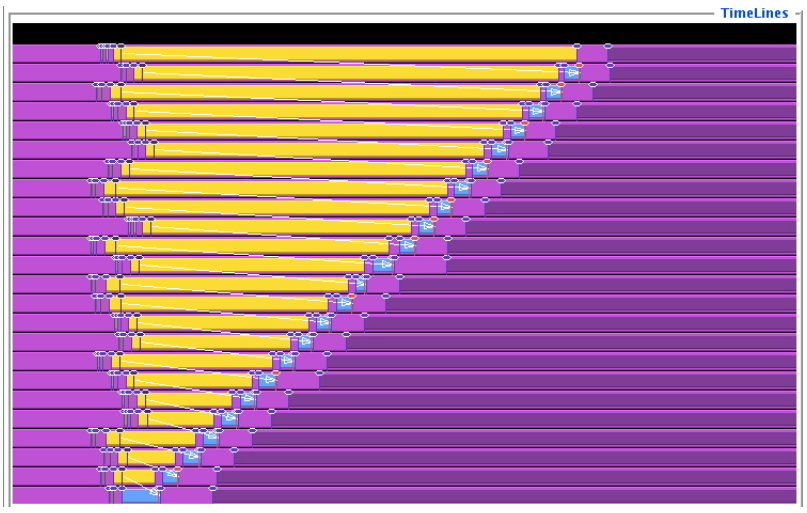


Figure 7.18 Trace of a simple send-recv code

Exercise 7.14 (Classroom exercise) Each student holds a piece of paper in the right hand – keep your left hand behind your back – and we want to execute:

1. Give the paper to your right neighbour;
2. Accept the paper from your left neighbour.

Including boundary conditions for first and last process, that becomes the following program:

1. If you are not the rightmost student, turn to the right and give the paper to your right neighbour.
2. If you are not the leftmost student, turn to your left and accept the paper from your left neighbour.

It is possible to orchestrate your processes to get an efficient and deadlock-free execution, but doing so is a bit cumbersome.

Exercise 7.15 The above solution treated every processor equally. Can you come up with a solution that uses blocking sends and receives, but does not suffer from the serialization behaviour?

A. Unix Introduction

Unix is an *Operating System (OS)*, that is, a layer of software between the user or a user program and the hardware. It takes care of files and screen output, and it makes sure that many processes can exist side by side on one system. However, it is not immediately visible to the user. Most of the time that you use Unix, you are typing commands which are executed by an interpreter called the *shell*. The shell makes the actual OS calls. There are a few possible Unix shells available, but in this tutorial we will assume that you are using the *sh* or *bash* shell, although many commands are common to the various shells in existence.

This short tutorial will get you going; if you want to learn more about Unix and shell scripting, see for instance `http://www.tldp.org/guides.html`. Most of this tutorial will work on any Unix-like platform, including *Cygwin* on Windows. However, there is not just one Unix:

- Traditionally there are a few major flavours of Unix: ATT and BSD. Apple has Darwin which is close to BSD; IBM and HP have their own versions of Unix, and Linux is yet another variant. The differences between these are deep down and if you are taking this tutorial you probably will not see them for quite a while.
- Within Linux there are various *Linux distributions* such as *Red Hat* or *Ubuntu*. These mainly differ in the organization of system files and again you probably need not worry about them.
- As mentioned just now, there are different shells, and they do differ considerably. Here you will learn the bash shell, which for a variety of reasons is to be preferred over the *csh* or *tcsh* shell. Other shells are the *ksh* and *zsh*.

For further reading:

- 'The Linux Command Line' by William Shotts `gd.tuwien.ac.at/linuxcommand`.

`org/` discusses life on the command line plus shell scripting.
- 'Bash Guide for Beginners' by Machtelt Garrels `tille.garrels.be/training/bash/` focuses on shell scripting in great detail.

A.1 Files and such

In this section you will learn about the Unix file system, which consists of *directories* that store *files*. You will learn about *executable* files and commands for displaying data files.

A.1.1 Looking at files

In this section you will learn commands for displaying file contents.

command	function
`ls`	list files or directories
`touch`	create new/empty file or update existing file
`cat > filename`	enter text into file
`cp`	copy files
`mv`	rename files
`rm`	remove files
`file`	report the type of file
`cat filename`	display file
`head,tail`	display part of a file
`less,more`	incrementally display a file

ls

Without any argument, the `ls` command gives you a listing of files that are in your present location.

Exercise A.1 Type `ls`. Does anything show up?

If there are files in your directory, they will be `ls`ted; if there are none, no output will be given. This is standard Unix behaviour: no output does not mean that something went wrong, it only means that there is nothing to report.

Exercise A.2 If the `ls` command shows that there are files, do `ls name` on one of those. By using an option, for instance `ls -s name` you can get more name.

If you specify a name of a non-existing file, you'll get an error message.

cat

The *cat* command is often used to display files, but it can also be used to create some simple content.

Exercise A.3 Type `cat > newfilename` (where you can pick any filename) and type some text. Conclude with `Control-d` on a line by itself[a]. Now use

`cat` to view the contents of that file: `cat newfilename`.

In the first use of `cat`, text was concatenated from the terminal to a file; in the second the file was cat'ed to the terminal output. You should see on your screen precisely what you typed into the file.

Be sure to type `Control-d` as the first thing on the last line of input. If you really get stuck, `Control-c` will usually get you out. Try this: start creating a file with `cat > filename` and hit `Control-c` in the middle of a line. What are the contents of your file?

[a] Press the `Control` and hold it while you press the `d` key.

Instead of `Control-d` you will often see the notation `^D`. The capital letter is for historic reasons: you use the control key and the lowercase letter.

The `ls` command can give you all sorts of information.

Exercise A.4 Read the man page of the `ls` command: `man ls`. Find out the size and the time/date date of the last change to some files, for instance the file you just created.

Did you find the `ls -s` and `ls -l` options? The first one lists the size of each file, usually in kilobytes, the other gives all sorts of information about a file, including things you will learn about later.

The `man` command puts you in a mode where you can view long text documents. This viewer is common on Unix systems (it is available as the `more` or `less` system command), so memorize the following ways of navigating: Use the space bar to go forward and the `u` key to go back up. Use `g` to go to the beginning fo the text, and `G` for the end. Use `q` to exit the viewer. If you really get stuck, `Control-c` will get you out.

There are several dates associated with a file, corresponding to changes in content, changes in permissions, and access of any sort. The *stat* command gives all of them.

If you already know what command you're looking for, you can use man to get online information about it. If you forget the name of a command, *man* `-k keyword` can help you find it.

touch

The *touch* command creates an empty file, or updates the timestamp of a file if it already exists. Use `ls -l` to confirm this behaviour.

cp, mv, rm

The *cp* can be used for copying a file (or directories, see below): `cp file1 file2` makes a copy of `file1` and names it `file2`.

Exercise A.5 Use `cp file1 file2` to copy a file. Confirm that the two files have the same contents. If you change the original, does anything happen to the copy?

You should see that the copy does not change if the original changes or is deleted.

If `file2` already exists, you will get an error message.

A file can be renamed with *mv*, for 'move'.

Exercise A.6 Rename a file. What happens if the target name already exists?

Files are deleted with `rm`. This command is dangerous: there is no undo.

head, tail

There are more commands for displaying a file, parts of a file, or information about a file.

Exercise A.7 Do `ls /usr/share/words` or `ls /usr/share/dict/words` to confirm that a file with words exists on your system. Now experiment with the commands `head`, `tail`, `more`, and `wc` using that file.

`head` displays the first couple of lines of a file, `tail` the last, and *more* uses the same viewer that is used for man pages. Read the man pages for these commands and experiment with increasing and decreasing the amount of output. The `wc` ('word count') command reports the number of words, characters, and lines in a file.

Another useful command is `file`: it tells you what type of file you are dealing with.

Exercise A.8 Do `file foo` for various 'foo': a text file, a directory, or the `/bin/ls` command.

Some of the information may not be intelligible to you, but the words to look out for are 'text', 'directory', or 'executable'.

At this point it is advisable to learn to use a text *editor*, such as *emacs* or *vi*.

A.1.2 Directories

Here you will learn about the Unix directory tree, how to manipulate it and how to move around in it.

command	function
`ls`	list the contents of directories
`mkdir`	make new directory
`cd`	change directory
`pwd`	display present working directory

A unix file system is a tree of directories, where a directory is a container for files or more directories. We will display directories as follows:

```
/ ..... The root of the directory tree
├── bin ..... Binary programs
└── home ..... Location of users directories
```

The root of the Unix directory tree is indicated with a slash. Do `ls /` to see what the files and directories there are in the root. Note that the root is not the location where you start when you reboot your personal machine, or when you log in to a server.

Exercise A.9 The command to find out your current working directory is `pwd`. Your home directory is your working directory immediately when you log in. Find out your home directory.

You will typically see something like `/home/yourname` or `/Users/yourname`. This is system dependent.

Do `ls` to see the contents of the working directory. In the displays in this section, directory names will be followed by a slash: `dir/` but this character is not part of their name. You can get this output by using `ls -F`, and you can tell your shell to use this output consistently by stating `alias ls=ls -F` at the start of your session. Example:

```
/home/you/
├── adirectory/
└── afile
```

The command for making a new directory is `mkdir`.

Exercise A.10 Make a new directory with `mkdir newdir` and view the current directory with `ls`.

You should see this structure:

```
/home/you/
└── newdir/ ..... the new directory
```

The command for going into another directory, that is, making it your working directory, is `cd` ('change directory'). It can be used in the following ways:

- `cd` Without any arguments, `cd` takes you to your home directory.
- `cd <absolute path>` An absolute path starts at the root of the directory tree, that is, starts with `/`. The `cd` command takes you to that location.
- `cd <relative path>` A relative path is one that does not start at the root. This form of the `cd` command takes you to `<yourcurrentdir>/<relative path>`.

Exercise A.11 Do `cd newdir` and find out where you are in the directory tree with `pwd`. Confirm with `ls` that the directory is empty. How would you get to this location using an absolute path?

`pwd` should tell you `/home/you/newdir`, and `ls` then has no output, meaning there is nothing to list. The absolute path is `/home/you/newdir`.

Exercise A.12 Let's quickly create a file in this directory: `touch onefile`, and another directory: `mkdir otherdir`. Do `ls` and confirm that there are a new file and directory.

You should now have:

```
/home/you/
└─ newdir/ ..... you are here
   ├─ onefile
   └─ otherdir/
```

The `ls` command has a very useful option: with `ls -a` you see your regular files and hidden files, which have a name that starts with a dot. Doing `ls -a` in your new directory should tell you that there are the following files:

```
/home/you/
└─ newdir/ ..... you are here
   ├─ .
   ├─ ..
   ├─ onefile
   └─ otherdir/
```

The single dot is the current directory, and the double dot is the directory one level back.

Exercise A.13 Predict where you will be after `cd ./otherdir/..` and check to see if you were right.

The single dot sends you to the current directory, so that does not change anything. The `otherdir` part makes that subdirectory your current working directory. Finally, `..` goes one level back. In other words, this command puts your right back where you started.

Since your home directory is a special place, there are shortcuts for `cd`'ing to it: `cd` without arguments, `cd ~`, and `cd $HOME` all get you back to your home.

Go to your home directory, and from there do `ls newdir` to check the contents of the first directory you created, without having to go there.

Exercise A.14 What does `ls ..` do?

Recall that `..` denotes the directory one level up in the tree: you should see your own home directory, plus the directories of any other users.

Exercise A.15 Can you use `ls` to see the contents of someone else's home directory? In the previous exercise you saw whether other users exist on your system. If so, do `ls ../thatotheruser`.

If this is your private computer, you can probably view the contents of the other user's directory. If this is a university computer or so, the other directory may very well be protected – permissions are discussed in the next section – and you get `ls: ../otheruser: Permission denied`.

Make an attempt to move into someone else's home directory with `cd`. Does it work?

You can make copies of a directory with `cp`, but you need to add a flag to indicate that you recursively copy the contents: `cp -r`. Make another directory `somedir` in your home so that you have

```
/home/you/
    newdir/ ..... you have been working in this one
    somedir/ ..... you just created this one
```

What is the difference between `cp -r newdir somedir` and `cp -r newdir thirddir` where `thirddir` is not an existing directory name?

A.1.3 Permissions

In this section you will learn about how to give various users on your system permission to do (or not to do) various things with your files.

Unix files, including directories, have permissions, indicating 'who can do what with this file'. Actions that can be performed on a file fall into three categories:

- reading `r`: any access to a file (displaying, getting information on it) that does not change the file;
- writing `w`: access to a file that changes its content, or even its metadata such as 'date modified';
- executing `x`: if the file is executable, to run it; if it is a directory, to enter it.

The people who can potentially access a file are divided into three classes too:

- the user `u`: the person owning the file;
- the group `g`: a group of users to which the owner belongs;
- other `o`: everyone else.

These nine permissions are rendered in sequence

user	*group*	*other*
rwx	*rwx*	*rwx*

For instance `rw-r-r-` means that the owner can read and write a file, the owner's group and everyone else can only read.

Permissions are also rendered numerically in groups of three bits, by letting `r` = 4, `w` = 2, `x` = 1:

rwx
$4+2+1$

Common codes are 7 = `rwx` and 6 = `rw`. You will find many files that have permissions 755 which stands for an executable that everyone can run, but only the owner can change, or 644 which stands for a data file that everyone can see but again only the owner can alter. You can set permissions by the `chmod` command:

```
chmod <permissions> file         # just one file
chmod -R <permissions> directory # directory, recursively
```

Examples:

```
chmod 766 file  # set to rwxrw-rw-
chmod g+w file  # give group write permission
chmod g=rx file # set group permissions
chod o-w  file  # take away write permission from others
chmod o=  file  # take away all permissions from others.
chmod g+r,o-x file # give group read permission
                   # remove other execute permission
```

The man page gives all options.

Exercise A.16 Make a file `foo` and do `chmod u-r foo`. Can you now inspect its contents? Make the file readable again, this time using a numeric code. Now make the file readable to your classmates. Check by having one of them read the contents.

When you've made the file 'unreadable' by yourself, you can still `ls` it, but not `cat` it: that will give a 'permission denied' message.

Make a file `com` with the following contents:

```
#!/bin/sh
echo "Hello world!"
```

This is a legitimate shell script. What happens when you type `./com`? Can you make the script executable?

In the three permission categories it is clear who 'you' and 'others' refer to. How about 'group'? We'll go into that in section A.5.

There are more obscure permissions. For instance the *setuid* bit declares that the program should run with the permissions of the creator, rather than the user executing it. This is useful for system utilities such `passwd` or `mkdir`, which alter the password file and the directory structure, for which *root priviliges* are needed. Thanks to the setuid bit, a user can run these programs, which are then so designed that a user can only make changes to their own password entry, and their own directories, respectively. The setuid bit is set with `chmod`: `chmod 4ugo file`.

A.1.4 Wildcards

You already saw that `ls filename` gives you information about that one file, and `ls` gives you all files in the current directory. To see files with certain conditions on their names, the *wildcard* mechanism exists. The following wildcards exist:

`*` any number of characters.

`?` any character.

Example:

```
%% ls
s       sk      ski     skiing  skill
%% ls ski*
ski     skiing  skill
```

The second option lists all files whose name start with `ski`, followed by any number of other characters'; below you will see that in different contexts `ski*` means '`sk` followed by any number of `i` characters'. Confusing, but that's the way it is.

A.2 Command execution

A.2.1 Search paths

In this section you will learn how Unix determines what to do when you type a command name.

If you type a command such as `ls`, the shell does not just rely on a list of commands: it will actually go searching for a program by the name `ls`. This means that you can have multiple different commands with the same name, and which one gets executed depends on which one is found first.

Exercise A.17 What you may think of as 'Unix commands' are often just executable files in a system directory. Do `which ls`, and do an `ls -l` on the result.

The location of `ls` is something like `/bin/ls`. If you `ls` that, you will see that it is probably owned by root. Its executable bits are probably set for all users.

The locations where unix searches for commands is the 'search path', which is stored in the *environment variable* (for more details see below) `PATH`.

Exercise A.18 Do `echo $PATH`. Can you find the location of `cd`? Are there other commands in the same location? Is the current directory '`.`' in the path? If not, do `export PATH=".:$PATH"`. Now create an executable file `cd` in the current director (see above for the basics), and do `cd`.

The path will be a list of colon-separated directories,
for instance `/usr/bin:/usr/local/bin:/usr/X11R6/bin`. If the working directory is in the path, it will probably be at the end: `/usr/X11R6/bin:.` but most likely it will not be there. If you put '`.`' at the start of the path, unix will find the local `cd` command before the system one.

Some people consider having the working directory in the path a security risk. If your directory is writable, someone could put a malicious script named `cd` (or any other system command) in your directory, and you would execute it unwittingly.

It is possible to define your own commands as aliases of existing commands.

Exercise A.19 Do `alias chdir=cd` and convince yourself that now `chdir` works just like `cd`. Do `alias rm='rm -i'`; look up the meaning of this in the man pages. Some people find this alias a good idea; can you see why?

The `-i` 'interactive' option for `rm` makes the command ask for confirmation before each delete. Since unix does not have a trashcan that needs to be emptied explicitly (as on Windows or the Mac OS), this can be a good idea.

A.2.2 Redirection

In this section you will learn how to feed one command into another, and how to connect commands to input and output files.

So far, the unix commands you have used have taken their input from your keyboard, or from a file named on the command line; their output went to your screen. There are other possibilities for providing input from a file, or for storing the output in a file.

Input redirection

The `grep` command had two arguments, the second being a file name. You can also write `grep string < yourfile`, where the less-than sign means that the input will come from the named file, `yourfile`. This is known as *input redirection*.

Output redirection

Conversely, `grep string yourfile > outfile` will take what normally goes to the terminal, and *redirect* the output to `outfile`. The output file is created if it didn't already exist, otherwise it is overwritten. (To append, use `grep text yourfile » outfile`.)

Exercise A.20 Take one of the grep commands from the previous section, and send its output to a file. Check that the contents of the file are identical to what appeared on your screen before. Search for a string that does not appear in the file and send the output to a file. What does this mean for the output file?

Searching for a string that does not occur in a file gives no terminal output. If you redirect the output of this `grep` to a file, it gives a zero size file. Check this with `ls` and `wc`.

Standard files

Unix has three standard files that handle input and output:

`stdin` is the file that provides input for processes.
`stdout` is the file where the output of a process is written.
`stderr` is the file where error output is written.

In an interactive session, all three files are connected to the user terminal. Using input or output redirection then means that the input is taken or the output sent to a different file than the terminal.

A.2.3 Command sequencing

There are various ways of having multiple commands on a single commandline.

Simple sequencing

First of all, you can type

```
command1 ; command2
```

This is convenient if you repeat the same two commands a number of times: you only need to up-arrow once to repeat them both.

There is a problem: if you type

```
gcc -o myprog myprog.c ; ./myprog
```

and the compilation fails, the program will still be executed, using an old version of the executable if that exists. This is very confusing.

A better way is:

```
gcc -o myprog myprog.c && ./myprog
```

which only executes the second command if the first one was successful.

Pipelining

Instead of taking input from a file, or sending output to a file, it is possible to connect two commands together, so that the second takes the output of the first as input. The syntax for this is `cmdone | cmdtwo`; this is called a pipeline. For instance, `grep a yourfile | grep b` finds all lines that contains both an `a` and a `b`.

Exercise A.21 Construct a pipeline that counts how many lines there are in your file that contain the string `th`. Use the `wc` command (see above) to do the counting.

Backquoting

There are a few more ways to combine commands. Suppose you want to present the result of `wc` a bit nicely. Type the following command

```
echo The line count is wc -l foo
```

where `foo` is the name of an existing file. The way to get the actual line count echoed is by the backquote:

```
echo The line count is ‘wc -l foo‘
```

Anything in between backquotes is executed before the rest of the command line is evaluated.

Exercise A.22 The way `wc` is used here, it prints the file name. Can you find a way to prevent that from happening?

A.2.4 Processes and jobs

`ps`	list (all) processes
`kill`	kill a process
`CTRL-c`	kill the foreground job
`CTRL-z`	suspect the foreground job
`jobs`	give the status of all jobs
`fg`	bring the last suspended job to the foreground
`fg %3`	bring a specific job to the foreground
`bg`	run the last suspended job in the background

The Unix operating system can run many programs at the same time, by rotating through the list and giving each only a fraction of a second to run each time. The command `ps` can tell you everything that is currently running.

Exercise A.23 Type `ps`. How many programs are currently running? By default `ps` gives you only programs that you explicitly started. Do `ps guwax` for a detailed list of everything that is running. How many programs are running? How many belong to the root user, how many to you?

To count the programs belonging to a user, pipe the `ps` command through an appropriate `grep`, which can then be piped to `wc`.

In this long listing of `ps`, the second column contains the *process numbers*. Sometimes it is useful to have those: if a program misbehaves you can `kill` it with

```
kill 123456
```

where `12345` is the process number.

To get dynamic information about all running processes, use the `top` command. Read the man page to find out how to sort the output by CPU usage.

Processes that are started in a shell are known as *jobs*. In addition to the process number, they have a job number. We will now explore manipulating jobs.

When you type a command and hit return, that command becomes, for the duration of its run, the *foreground process*. Everything else that is running at the same time is a *background process*.

Make an executable file `hello` with the following contents:

```
#!/bin/sh
while [ 1 ] ; do
  sleep 2
  date
done
```

and type `./hello`.

Exercise A.24 Type `Control-z`. This suspends the foreground process. It will give you a number like `[1]` or `[2]` indicating that it is the first or second program that has been suspended or put in the background. Now type `bg` to put this process in the background. Confirm that there is no foreground process by hitting return, and doing an `ls`.

After you put a process in the background, the terminal is available again to accept foreground commands. If you hit return, you should see the command prompt. However, the background process still keeps generating output.

Exercise A.25 Type `jobs` to see the processes in the current session. If the process you just put in the background was number 1, type `fg %1`. Confirm that it is a foreground process again.

If a shell is executing a program in the foreground, it will not accept command input, so hitting return should only produce blank lines.

Exercise A.26 When you have made the `hello` script a foreground process again, you can kill it with `Control-c`. Try this. Start the script up again, this time as `./hello &` which immediately puts it in the background. You should also get output along the lines of `[1] 12345` which tells you that it is the first job you put in the background, and that `12345` is its process ID. Kill the script with `kill %1`. Start it up again, and kill it by using the process number.

The command `kill 12345` using the process number is usually enough to kill a running program. Sometimes it is necessary to use `kill -9 12345`.

A.3 Shell environment variables

Above you encountered `PATH`, which is an example of an shell, or environment, variable. These are variables that are known to the shell and that can be used by all programs run by the shell. You can see the full list of all variables known to the shell by typing `env`.

You can get the value of a shell variable by prefixing it with a dollar sign. Type the following two commands and compare the output:

```
echo PATH
echo $PATH
```

Exercise A.27 Check on the value of the `HOME` variable by typing `echo $HOME`. Also find the value of `HOME` by piping `env` through `grep`.

Environment variables can be set in a number of ways. The simplest is by an assignment as in other programming languages.

Exercise A.28 Type `a=5` on the commandline. This defines a variable `a`; check on its value by using the `echo` command.

The shell will respond by typing the value `5`.

Beware not to have space around the equals sign; also be sure to use the dollar sign to print the value.

A variable set this way will be known to all subsequent commands you issue in this shell, but not to commands in new shells you start up. For that you need the `export` command. Reproduce the following session (the square brackets form the command prompt):

```
[] a=20
[] echo $a
20
[] /bin/bash
[] echo $a

[] exit
exit
[] export a=21
[] /bin/bash
[] echo $a
21
[] exit
```

A.4 Shell interaction

Interactive use of Unix, in contrast to script writing, is a complicated conversation between the user and the shell. You, the user, type a line, hit return, and the shell tries to interpret it. There are several cases.

- Your line contains one full command, such as `ls foo`: the shell will execute this command.
- You can put more than one command on a line, separated by semicolons: `mkdir foo; cd foo`. The shell will execute these commands in sequence.
- Your input line is not a full command, for instance `while [1]`. The shell will recognize that there is more to come, and use a different prompt to show you that it is waiting for the remainder of the command.
- Your input line would be a legitimate command, but you want to type more on a second line. In that case you can end your input line with a backslash character (\), and the shell will recognize that it needs to hold off on executing your command. In effect, the backslash will hide (*escape*) the return.

When the shell has collected a command line to execute, by using one or more of your input line or only part of one, as described just now, it will apply expansion to the command line. It will then interpret the commandline as a command and arguments, and proceed to invoke that command with the

arguments as found.

There are some subtleties here. If you type `ls *.c`, then the shell will reognize the wildcard character and expand it to a command line, for instance `ls foo.c bar.c`. Then it will invoke the `ls` command with the argument list `foo.c bar.c`. Note that `ls` does not receive `*.c` as argument! In cases where you do want the unix command to receive an argument with a wildcard, you need to escape it so that the shell will not expand it. For instance, `find . -name *.c` will make the shell invoke `find` with arguments `. -name *.c`.

A.5 The system and other users

Unix is a multi-user operating system. Thus, even if you use it on your own personal machine, you are a user with an *account* and you may occasionally have to type in your username and password.

If you are on your personal machine, you may be the only user logged in. On university machines or other servers, there will often be other users. Here are some commands relating to them.

`whoami` show your login name.

`who` show the other users currently logged in.

`finger otheruser` get information about another user; you can specify a user's login name here, or their real name, or other identifying information the system knows about.

`top` which processes are running on the system; use `top -u` to get this sorted the amount of cpu time they are currently taking. (On Linux, try also the `vmstat` command.)

`uptime` how long has it been since your last reboot?

A.5.1 Groups

In section A.1.3 you saw that there is a permissions category for 'group'. This allows you to open up files to your close collaborators, while leaving them protected from the wide world.

When your account is created, your system administrator will have assigned you to one or more groups. (If you admin your own machine, you'll be in some default group; read on for adding yourself to more groups.)

The command `groups` tells you all the groups you are in, and `ls -l` tells you what group a file belongs to. Analogous to `chmod`, you can use `chgrp` to change the group to which a file belongs, to share it with a user who is also in that group.

Creating a new group, or adding a user to a group needs system priviliges. To create a group:

```
sudo groupadd new_group_name
```

To add a user to a group:

```
sudo usermod -a -G thegroup theuser
```

A.5.2 The super user

Even if you own your machine, there are good reasons to work as much as possible from a regular user account, and use *root priviliges* only when strictly needed. (The root account is also known as the *super user*.) If you have root priviliges, you can also use that to 'become another user' and do things with their priviliges.

- To execute a command as another user:

  ```
  sudo -u otheruser command arguments
  ```

- To execute a command as the root user:

  ```
  sudo command arguments
  ```

- Become another user:

  ```
  sudo su - otheruser
  ```

- Become the *super user*:

  ```
  sudo su -
  ```

B. Storing Fractions

In contrast to storing integers, the storage of a value with a fractional part requires that we store not only the pattern of 0s and 1s representing its binary representation, but also the position of the radix point. A popular way of doing this is based on scientific notation and is called **floating-point** notation.

Floating-Point Notation

Let us explain floating-point notation with an example using only one byte of storage. Although machines normally use much longer patters, the 8-bit format is representative of actual systems and serves to demonstrate the important concepts without the clutter of long bit patterns.

We first designate the high-order bit of the byte as the sign-bit. A 0 in the sign bit will mean that the value stored is nonnegative, and a 1 will mean that the value is negative. Next we divide the remaining 7 bits of the byte into two groups, or fields: the **exponent field** and the **mantissa field**. Let us designate 3 bites following the sign bit as the exponent field and the remaining 4 bits as the mantissa field. Figure B.1 illustrates how the byte is divided.

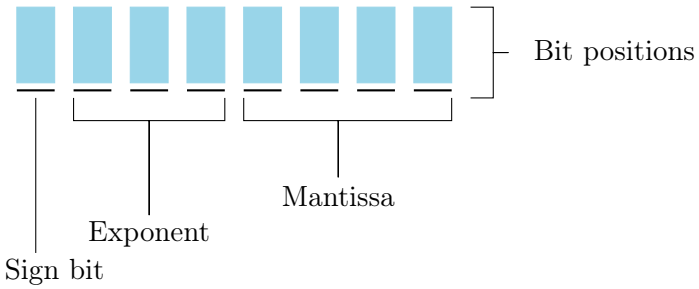


Figure B.1 Floating-point notation components

We can explain the meaning of the fields by considering the following

example. Suppose a byte consisting of the bit pattern 01101011. Analysing this pattern with the preceding format, we see that the sign bit is 0, the exponent is 110, and the mantissa is 1011. To decode, we first extract the mantissa and place a radix point to its left side, obtaining:

```
.1011
```

Next, we extract the contents of the exponent field (110) and interpret it as an integer storing stored using 3-bit excess method. Thus the pattern in the exponent field in our example represents a positive 2. This tells us to move the radix in our solution to the right by 2 (a negative exponent would mean to move the radix to the left). Consequently, we obtain

```
10.11
```

which is the binary representation of $2\frac{1}{4}$. Next, we note that the sign bit in our example is 0: the value represented is thus nonnegative. We conclude that the byte 01101011 represents $2\frac{1}{4}$. Had the pattern been 11101011 (which is the same as before except for the sign bit), the value represented would have been $-2\frac{1}{4}$.

As another example, consider the byte 00111100. We extract the mantissa to obtain

```
.1100
```

and move the radix by 1 bit to the left, since the exponent field (011) represents the value -1. We therefore have

```
.01100
```

which represents $\frac{3}{8}$. Since the sign bit in the original pattern is 0, the value stored is nonnegative. We conclude that the pattern 00111100 represents $\frac{3}{8}$.

To store a value using floating-point notation, we reverse the preceding process. For example, to encode $1\frac{1}{8}$, first we express it in binary notation and obtain 1.001. Next we copy the bit pattern into the mantissa field from left to right, starting with the leftmost 1 in the binary representation. At this point the byte looks like this:

```
_ _ _ _ 1 0 0 1
```

We must now fill i nthe exponent field. To this end, we imagine the contents of the mantissa field with a radix point at its left and determine the number of bits and the direction the radix must be moved be moved to obtain the original binary number. In our example, we see that the radix in .1001 must be moved 1 bit to the right to obtain 1.001. The exponent should therefore be a positive one, so we place 101 (which is positive one in excess notation) in the exponent field. Finally, we fill the sign bit with 0 because the value being stored is nonnegative. The finished byte looks like this:

```
0 1 0 1 1 0 0 1
```

There is a subtle point you may have missed when filling in the mantissa field. The rule is to copu the bit pattern appearing in the binary representation from left to right, starting with the leftmost 1. To clarify, consider the process of storing the value $\frac{3}{8}$, which is .011 in binary notation. In this case the mantissa will be:

_ _ _ _ 1 1 0 0

it will not be

_ _ _ _ 0 1 1 0

This is because we fill the mantissa field *starting with the leftmost 1* that appears in the binary representation. Representations that conform to this rule are said to be in **normalised form**.

Using normalised form eliminates the possibility of multiple representations for the same value. For example, both 00111100 and 01000110 would decode to the value $\frac{3}{8}$, but only the first pattern is in normalised form. Complying with normalised form also means that the representation for all nonzero values will have a mantissa, that starts with 1. The value zero, however, is a special case; its floating-point representation is a bit pattern of all 0s.

Truncation Errors

Let us consider the annoying problem that occurs if we try to store the value $2\frac{5}{8}$ with our one-byte floating-point system. We first write $2\frac{5}{8}$ in binary, which gives us 10.101. But when we copy this into the mantissa field, we run out of room, and the rightmost 1 (which represents the last $\frac{1}{8}$) is lost (see Figure B.2). If we ignore this problem for now and continue by filling in the exponent field and the sign bit, we end up with the bit pattern 01101010, which represents $2\frac{1}{2}$ instead of $2\frac{5}{8}$. What has occurred is called a **truncate error**, or **round-off error** – meaning that part of the value being stored is lost because the mantissa field is not large enough.

THe significance of such errors can be reduced by using a longer mantissa field. In fact, most computers manufactured today use at least 32 bits for storing values in floating-point notation instead of the 8 bits we have used here. This also allows for a longer exponent field at the same time. Even with these longer formats, however, there are still times when more accuracy is required.

Another source of truncation errors is a phenomenon that you are already accustomed to in base ten notation: the problem of nonterminating expansions, such as those found when trying to express $\frac{1}{3}$ in decimal form. Some values cannot be accurately expressed regardless of how many digits we use. The difference between our traditional base ten notation and binary notation is that more values have nonterminating representations in binary notation than in decimal notation. For example, the value one-tenth is nonterminating when expressed in binary. Imagine the problems this might cause the unwary person using floating-point notation to store and manipulate dollars and cents. In

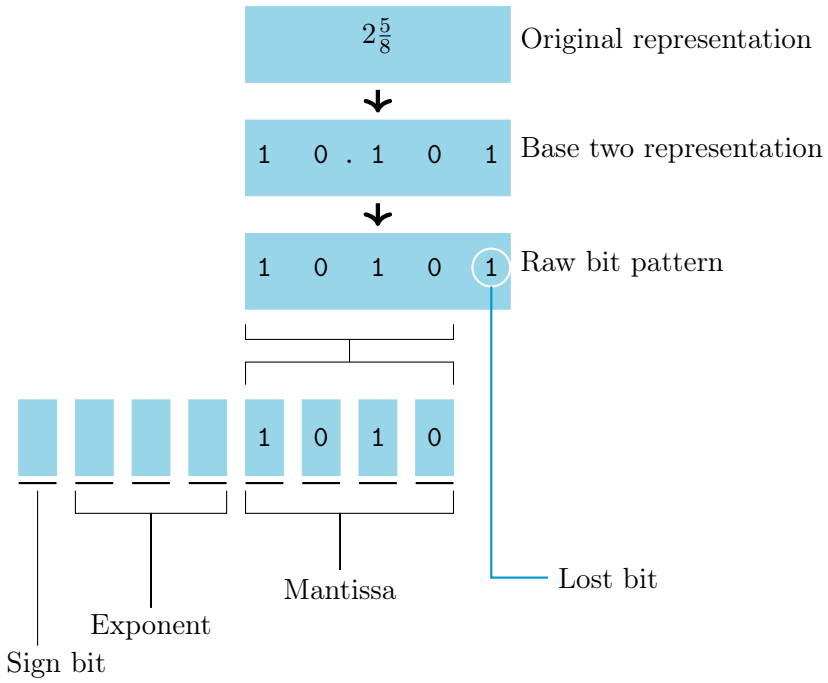


Figure B.2 Floating-point notation components

particular, if the dollar is used as the unit of measure, the value of a dime could not be stored accurately. A solution in this case is to manipulate the data in units of pennies so that all values are integers that can be accurately stored using a method such as two's complement.

Truncation errors and their related problems are an everyday concern for people working in the area of numerical analysis. This branch of mathematics deals with the problems involved when doing actual computations that are often massive and require significant accuracy.

The following is an example that would warm the heart of any numerical analyst. Suppose we are asked to add the following three values using our one-byte floating-point notation defined precviously:

$$2\tfrac{1}{2} + \tfrac{1}{8} + \tfrac{1}{8}$$

If we add the values in the order listed, we first add $2\frac{1}{2}$ to $\frac{1}{8}$ and obtain $2\frac{5}{8}$ which in binary is 10.101. Unfortunately, because this value cannot be stored accurately (as seen previously), the result of our first step ends up being stored as $2\frac{1}{2}$ (which is the same as one of the values we were adding). The next step is to add this result to the last $\frac{1}{8}$ Here again a truncation error occurs, and our final result turns out to be the incorrect answer $2\frac{1}{2}$.

Now let us add the values in the opposite order. We first add $\frac{1}{8}$ to $\frac{1}{8}$ to obtain $\frac{1}{4}$. In binary this is .01; so the result of our first step is stored in a byte as 00111000, which is accurate. We now add this $\frac{1}{4}$ to the next value in the list, $2\frac{1}{2}$, and obtain $2\frac{3}{4}$, which we can accurately store in a byte as 01101011. THe result this time is the correct answer.

To summarise, in adding numerica values represented in floating-point

notation, the order in which they are added can be important. The problem is that if a very large number is added to a very small number, the small number may be truncated. Thus, the general rule for adding multiple values values is to add the smaller values first, in hopes that they will accumulate to a value that is significant when adding to the larger values. This was the phenomenon experienced in the preceding example.

Designers of today's commercial software packages do a good job of shielding the uneducated user from problems such as this. In a typical spreadsheet system, correct answers will be obtained unless the values being added differ in size by a factor of 10^{16} or more. Thus, if you found it necessary to add one to the value

$$10,000,000,000,000,000$$

you might get the answer

$$10,000,000,000,000,000$$

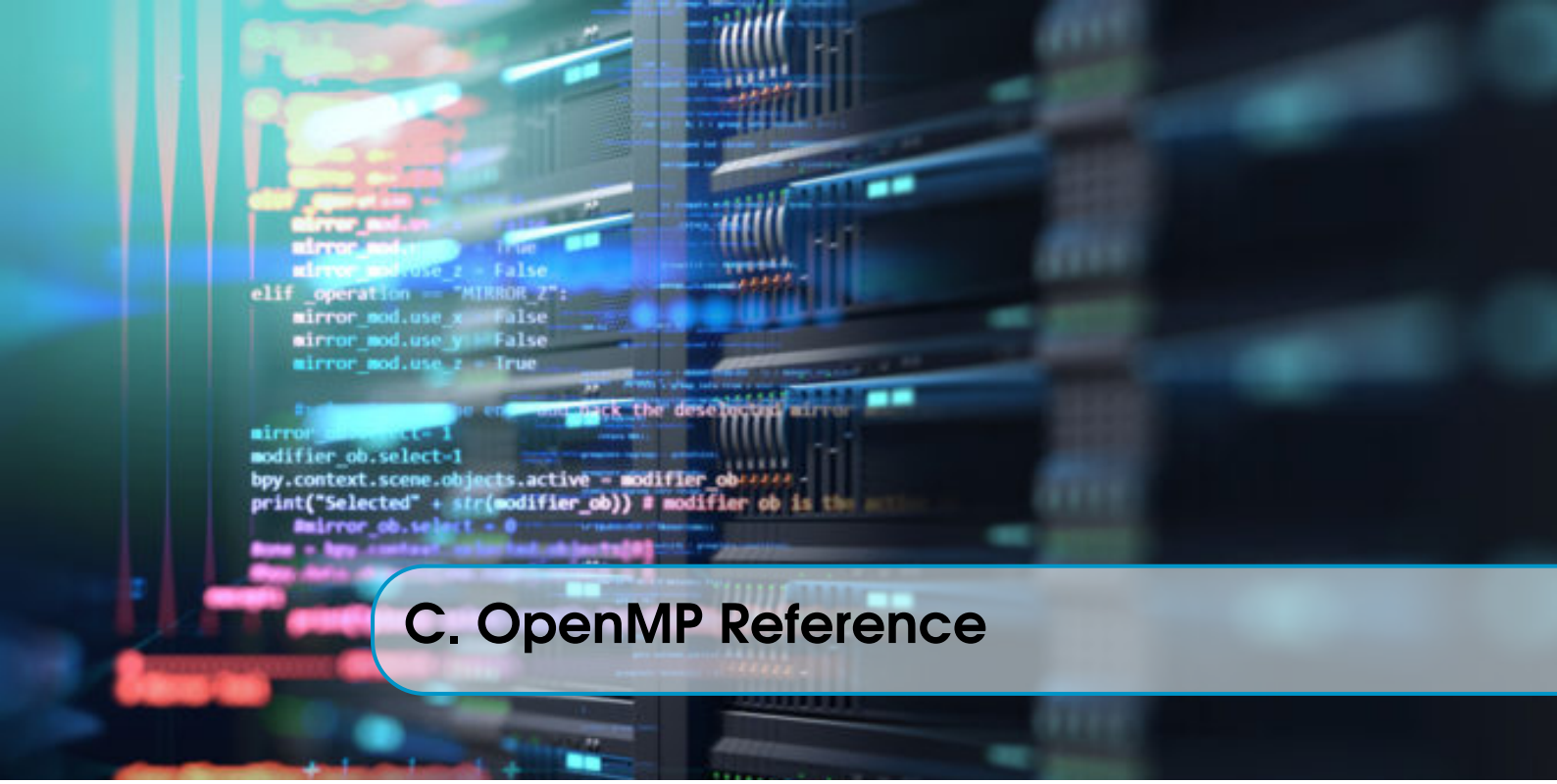
rather than

$$10,000,000,000,000,001$$

Such problems are significant in applications (such as navigational systems) in which minor errors can be compounded in additional computations and ultimately produce significant consequences, but for the typical PC user the degree of accuracy offered by most commercial software is sufficient.

Aside B.1 — Single Precision Floating Point.
The floating-point notation introduced in this chapter is far too simplistic to be used in an actual computer. After all, with just 8 bits only 256 numbers out of the set of all real numbers can be expressed. Our discussion has used 8 bits to keep the examples simple, yet still cover the important underlying concepts.

Many of today's computer support a 32 bit form of this notation called **Single Precision Floating Point**. This format uses 1 bit for the sign, 8 bits for the exponent (in an excess notation), and 23 bits for the mantissa. Thus, single precision floating point is capable of expressing very large numbers (order of 10^{38}) down to very small numbers (order of 10^{-37}) with the precision of 7 decimal digits. That is to say, the first 7 digits of a given decimal number can be stored with very good accurary (a small amount of error may still be present). Any digits past the first 7 will certainly be lost by truncation error (although the magnitude of the number is retained). Another form, called **Double Precision Floating Point**, uses 64 bits and provides a precision of 15 decimal digits.



C. OpenMP Reference

C.1 Runtime functions and internal control variables

OpenMP has a number of settings that can be set through *environment variables*, and both queried and set through *library routines*. These settings are called *Internal Control Variables (ICV)*: an OpenMP implementation behaves as if there is an internal variable storing this setting.

The runtime functions are:

- `omp_set_num_threads`
- `omp_get_num_threads`
- `omp_get_max_threads`
- `omp_get_thread_num`
- `omp_get_num_procs`
- `omp_in_parallel`
- `omp_set_dynamic`
- `omp_get_dynamic`
- `omp_set_nested`
- `omp_get_nested`
- `omp_get_wtime`
- `omp_get_wtick`
- `omp_set_schedule`
- `omp_get_schedule`
- `omp_set_max_active_levels`
- `omp_get_max_active_levels`
- `omp_get_thread_limit`
- `omp_get_level`
- `omp_get_active_level`
- `omp_get_ancestor_thread_num`
- `omp_get_team_size`

Here are the OpenMP *environment variables*:

- `OMP_CANCELLATION` Set whether cancellation is activated
- `OMP_DISPLAY_ENV` Show OpenMP version and environment variables
- `OMP_DEFAULT_DEVICE` Set the device used in target regions
- `OMP_DYNAMIC` Dynamic adjustment of threads
- `OMP_MAX_ACTIVE_LEVELS` Set the maximum number of nested parallel regions
- `OMP_MAX_TASK_PRIORITY` Set the maximum task priority value
- `OMP_NESTED` Nested parallel regions
- `OMP_NUM_THREADS` Specifies the number of threads to use
- `OMP_PROC_BIND` Whether theads may be moved between CPUs
- `OMP_PLACES` Specifies on which CPUs the theads should be placed
- `OMP_STACKSIZE` Set default thread stack size
- `OMP_SCHEDULE` How threads are scheduled
- `OMP_THREAD_LIMIT` Set the maximum number of threads
- `OMP_WAIT_POLICY` How waiting threads are handled

There are 4 ICV that behave as if each thread has its own copy of them. The default is implementation-defined unless otherwise noted.

- It may be possible to adjust dynamically the number of threads for a parallel region. Variable: `OMP_DYNAMIC`; routines: `omp_set_dynamic`, `omp_get_dynamic`.
- If a code contains *nested parallel regions*, the inner regions may create new teams, or they may be executed by the single thread that encounters them. Variable: `OMP_NESTED`; routines `omp_set_nested`, `omp_get_nested`. Allowed values are `TRUE` and `FALSE`; the default is false.
- The number of threads used for an encountered parallel region can be controlled. Variable: `OMP_NUM_THREADS`; routines `omp_set_num_threads`, `omp_get_max_threads`.
- The schedule for a parallel loop can be set. Variable: `OMP_SCHEDULE`; routines `omp_set_schedule`, `omp_get_schedule`.

Non-obvious syntax:

```
export OMP_SCHEDULE="static,100"
```

Other settings:

- `omp_get_num_threads`: query the number of threads active at the current place in the code; this can be lower than what was set with `omp_set_num_threads`. For a meaningful answer, this should be done in a parallel region.
- `omp_get_thread_num`
- `omp_in_parallel`: test if you are in a parallel region (see for instance section 6.4).
- `omp_get_num_procs`: query the physical number of cores available.

Other environment variables:

- `OMP_STACKSIZE` controls the amount of space that is allocated as per-thread *stack*; the space for private variables.
- `OMP_WAIT_POLICY` determines the behaviour of threads that wait, for instance for *critical section*:

  - `ACTIVE` puts the thread in a *spin-lock*, where it actively checks whether it can continue;
  - `PASSIVE` puts the thread to sleep until the OS wakes it up.

  The 'active' strategy uses CPU while the thread is waiting; on the other hand, activating it after the wait is instantaneous. With the 'passive' strategy, the thread does not use any CPU while waiting, but activating it again is expensive. Thus, the passive strategy only makes sense if threads will be waiting for a (relatively) long time.
- `OMP_PROC_BIND` with values `TRUE` and `FALSE` can bind threads to a processor. On the one hand, doing so can minimize data movement; on the other hand, it may increase load imbalance.

C.2 Timing

OpenMP has a wall clock timer routine `omp_get_wtime`

```
double omp_get_wtime(void);
```

The starting point is arbitrary and is different for each program run; however, in one run it is identical for all threads. This timer has a resolution given by `omp_get_wtick`.

Exercise C.1 Use the timing routines to demonstrate speedup from using multiple threads.

- Write a code segment that takes a measurable amount of time, that is, it should take a multiple of the tick time.
- Write a parallel loop and measure the speedup. You can for instance do this

```
for (int use_threads=1; use_threads<=nthreads; use_threads++) {
#pragma omp parallel for num_threads(use_threads)
    for (int i=0; i<nthreads; i++) {
        .....
    }
    if (use_threads==1)
      time1 = tend-tstart;
    else // compute speedup
```

- In order to prevent the compiler from optimizing your loop away, let the body compute a result and use a reduction to preserve these results.

C.3 Thread safety

With OpenMP it is relatively easy to take existing code and make it parallel by introducing parallel sections. If you're careful to declare the appropriate variables shared and private, this may work fine. However, your code may include calls to library routines that include a *race condition*; such code is said not to be *thread-safe*.

For example a routine

```
static int isave;
int next_one() {
 int i = isave;
 isave += 1;
 return i;
}

...
for ( .... ) {
  int ivalue = next_one();
}
```

has a clear race condition, as the iterations of the loop may get different `next_one` values, as they are supposed to, or not. This can be solved by using an *critical* pragma for the `next_one` call; another solution is to use an *threadprivate* declaration for `isave`. This is for instance the right solution if the `next_one` routine implements a *random number generator*.

C.4 Performance and tuning

The performance of an OpenMP code can be influenced by the following.

Amdahl effects Your code needs to have enough parts that are parallel (see 4.2.3). Sequential parts may be sped up by having them executed redundantly on each thread, since that keeps data locally.

Dynamism Creating a thread team takes time. In practice, a team is not created and deleted for each parallel region, but creating teams of different sizes, or recursize thread creation, may introduce overhead.

Load imbalance Even if your program is parallel, you need to worry about load balance. In the case of a parallel loop you can set the *schedule* clause to `dynamic`, which evens out the work, but may cause increased communication.

Communication Cache coherence causes communication. Threads should, as much as possible, refer to their own data.

- Threads are likely to read from each other's data. That is largely unavoidable.
- Threads writing to each other's data should be avoided: it may require synchronization, and it causes coherence traffic.
- If threads can migrate, data that was local at one time is no longer local after migration.
- Reading data from one socket that was allocated on another socket is inefficient.

Affinity Both data and execution threads can be bound to a specific locale to some extent. Using local data is more efficient than remote data, so you want to use local data, and minimize the extent to which data or execution can move.

- See the above points about phenomena that cause communication.
- It is possible to bind threads to places. There can, but does not need, to be an effect on affinity. For instance, if an OpenMP thread can migrate between hardware threads, cached data will stay local. Leaving an OpenMP thread completely free to migrate can be advantageous for load balancing, but you should only do that if data affinity is of lesser importance.
- Static loop schedules have a higher chance of using data that has affinity with the place of execution, but they are worse for load balancing. On the other hand, the `nowait` clause can aleviate some of the problems with static loop schedules.

Binding You can choose to put OpenMP threads close together or to spread them apart. Having them close together makes sense if they use lots of shared data. Spreading them apart may increase bandwidth.

Synchronization Barriers are a form of synchronization. They are expensive by themselves, and they expose load imbalance. Implicit barriers happen at the end of worksharing constructs; they can be removed with `nowait`. Critical sections imply a loss of parallelism, but they are also slow as they are realized through *operating system* functions. These are often quite costly, taking many thousands of cycles. Critical sections should be used only if the parallel work far outweighs it.

C.5 Accelerators

In OpenMP 4.0 there is support for offloading work to an *accelerator* or *co-processor*:

```
#pragma omp target [clauses]
```

with clauses such as

- `data`: place data
- `update`: make data consistent between host and device

D. MPI Routines

D.1 How to read routine prototypes

Throughout the MPI part of this book we will refer to various MPI routines. In this chapter we give the reference syntax of the routines. These typically comprise:

- The semantics: routine name and list of parameters and what they mean.
- C synxtax: the routine definition as it appears in the *mpi.h* file.
- Python syntax: routine name, indicating to what class it applies, and parameter, indicating which ones are optional.

These 'routine prototypes' look like code but they are not! Here is how you translate them.

C

The typically C routine specification in MPI looks like:

```
int MPI_Comm_size(MPI_Comm comm,int *nprocs)
```

This means that

- The routine returns an `int` parameter. Strictly speaking you should test against `MPI_SUCCESS`:

```
MPI_Comm comm = MPI_COMM_WORLD;
int nprocs;
int errorcode;
errorcode = MPI_Comm_world( MPI_COMM_WORLD,&nprocs);
if (errorcode!=MPI_SUCCESS) {
  printf("Routine MPI_Comm_world failed! code=%d\cpp",
         errorcode);
  return 1;
}
```

However, the error codes are hardly ever useful, and there is not much your program can do to recover from an error. Most people call the routine as

```
MPI_Comm_world( /* parameter ... */ );
```

- The first argument is of type `MPI_Comm`. This is not a C built-in datatype, but it behaves like one. There are many of these `MPI_something` datatypes in MPI. So you can write:

```
MPI_Comm my_comm =
    MPI_COMM_WORLD; // using a predefined value
MPI_Comm_size( comm, /* remaining parameters */ );
```

- Finally, there is a 'star' parameter. This means that the routine wants an address, rather than a value. You would typically write:

```
MPI_Comm my_comm = MPI_COMM_WORLD; // using a predef. value
int nprocs;
MPI_Comm_size( comm, &nprocs );
```

Seeing a 'star' parameter usually means either: the routine has an array argument, or: the routine internally sets the value of a variable. The latter is the case here.

Python

The Python interface to MPI uses classes and objects. Thus, a specification like:

```
MPI.Comm.Send(self, buf, int dest, int tag=0)
```

should be parsed as follows.

- First of all, you need the `MPI` class:

```
from mpi4py import MPI
```

- Next, you need a `Comm` object. Often you will use the predefined communicator

```
comm = MPI.COMM_WORLD
```

- The keyword `self` indicates that the actual routine `Send` is a method of the `Comm` object, so you call:

```
comm.Send( .... )
```

- Parameters that are listed by themselves, such as `buf`, as positional. Parameters that are listed with a type, such as `int dest` are keyword parameters. Keyword parameters that have a value specified, such as `int tag=0` are optional, with the default value indicated. Thus, the typicall call for this routine is:

```
comm.Send(sendbuf,dest=other)
```

specifying the send buffer as positional parameter, the destination as keyword parameter, and using the default value for the optional tag.

Some python routines are 'class methods', and their specification lacks the `self` keyword. For instance:

```
MPI.Request.Waitall(type cls, requests, statuses=None)
```

would be used as

```
MPI.Request.Waitall(requests)
```

D.2 MPI Routines

There are many more MPI routines than the ones listed here. See `https://www.mpich.org/static/docs/latest/www3/index.htm` for more details about these routines and others.

MPI_Init

```
C:
int MPI\_Init(int *argc, char ***argv)
```

Python does not require a separate init-statement.

MPI_Finalize

```
C:
int MPI_Finalize(void)
```

Python does not require a separate finalize-statement.

MPI_Initialized

```
C:
int MPI_Initialized(int *flag)

Parameters:
  OUT flag: true if MPI_Initialize has been called;
            false otherwise
```

MPI_Finalized

```
C:
int MPI_Finalized(int *flag)

Parameters:
  OUT flag: true if MPI_Finalize has been called;
            false otherwise
```

MPI_Get_processor_name

```
C:
int MPI_Get_processor_name(char *name, int *resultlen)

Parameters:
  OUT name: buffer char[MPI_MAX_PROCESSOR_NAME]
  OUT resultlen: length of name (integer)

Python:
MPI.Get_processor_name()
```

MPI_Comm_size

```
C:
int MPI_Comm_size(MPI_Comm comm, int *size)

Parameters:
MPI_COMM_SIZE(comm, size)
  IN comm: communicator (handle)
  OUT size: number of processes in the group of comm (int)

Python:
MPI.Comm.Get_size(self)
```

MPI_Comm_rank

```
C:
int MPI_Comm_rank(MPI_Comm comm, int *rank)

Parameters:
MPI_COMM_RANK(comm, rank)
  IN comm: communicator (handle)
  OUT rank: rank of calling process in group of comm (int)

Python:
MPI.Comm.Get_rank(self)
```

MPI_Allreduce

```
C:
int MPI_Allreduce(const void* sendbuf, void* recvbuf,
   int count, MPI_Datatype datatype, MPI_Op op,
   MPI_Comm comm)

Parameters:
```

```
  IN sendbuf: starting address of send buffer (choice)
  OUT recvbuf: starting address of receive buffer (choice)
  IN count: number of elements in send buffer (non-neg. int)
  IN datatype: data type of elements of send buffer (handle)
  IN op: operation (handle)
  IN comm: communicator (handle)

Python native:
recvobj = MPI.Comm.allreduce(self, sendobj, op=SUM)
Python numpy:
MPI.Comm.Allreduce(self, sendbuf, recvbuf, Op op=SUM)
```

MPI_Reduce

```
C:
int MPI_Reduce(
  const void* sendbuf, void* recvbuf, int count,
  MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Parameters:
  IN sendbuf: starting address of send buffer (choice)
  OUT recvbuf: starting address of receive buffer (choice)
  IN count: number of elements in send buffer (non-neg. int)
  IN datatype: data type of elements of send buffer (handle)
  IN op: operation (handle)
  IN root: rank of receiving process (integer)
  IN comm: communicator (handle)

Python native:
comm.reduce(self, sendobj=None, recvobj=None, op=SUM,
    int root=0)
Python numpy:
comm.Reduce(self, sendbuf, recvbuf, Op op=SUM, int root=0)
```

MPI_Bcast

```
C:
int MPI_Bcast(
  void* buffer, int count, MPI_Datatype datatype,
  int root, MPI_Comm comm)

Parameters:
  IN buffer: starting address of buffer (choice)
  IN count: number of elements in buffer (non-neg. int)
```

```
  IN datatype: data type of elements in buffer (handle)
  IN comm: communicator (handle)

Python native:
rbuf = MPI.Comm.bcast(self, obj=None, int root=0)
Python numpy:
MPI.Comm.Bcast(self, buf, int root=0)
```

MPI_Gather

```
C:
int MPI_Gather(
  const void* sendbuf, int sendcount, MPI_Datatype sendtype,
  void* recvbuf, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm)

Parameters:
  IN sendbuf: starting address of send buffer (choice)
  IN sendcount: number of elements in send buffer (non-neg. int)
  IN sendtype: data type of send buffer elements (handle)
  OUT recvbuf: address of receive buffer
     (choice, significant only at root)
  IN recvcount: number of elements for any single receive
     (non-negative integer, significant only at root)
  IN recvtype: data type of recv buffer elements
     (significant only at root) (handle)
  IN root: rank of receiving process (integer)
  IN comm: communicator (handle)

Python:
MPI.Comm.Gather(self, sendbuf, recvbuf, int root=0)
```

MPI_Allgather

```
C:
int MPI_Allgather(const void *sendbuf, int  sendcount,
     MPI_Datatype sendtype, void *recvbuf, int recvcount,
     MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Iallgather(const void *sendbuf, int  sendcount,
     MPI_Datatype sendtype, void *recvbuf, int recvcount,
     MPI_Datatype recvtype, MPI_Comm comm,
     MPI_Request *request)

Parameters:
```

```
  IN sendbuf : Starting address of send buffer (choice)
  IN sendcount: Number of elements in send buffer (integer)
  IN sendtype: Datatype of send buffer elements (handle)
  OUT recvbuf: Starting address of recv buffer (choice)
  OUT recvcount: Number of elements received from any
      process (int)
  OUT recvtype: Datatype of receive buffer elements (handle)
  IN comm; Communicator (handle)
  OUT request: Request (handle, non-blocking only)
```

MPI_Send

```
C:
int MPI_Send(
  const void* buf, int count, MPI_Datatype datatype,
  int dest, int tag, MPI_Comm comm)

Parameters:
  IN buf: initial address of send buffer (choice)
  IN count: number of elements in send buffer (non-neg. int)
  IN datatype: datatype of each send buffer element (handle)
  IN dest: rank of destination (integer)
  IN tag: message tag (integer)
  IN comm: communicator (handle)

Python native:
MPI.Comm.send(self, obj, int dest, int tag=0)
Python numpy:
MPI.Comm.Send(self, buf, int dest, int tag=0)
```

MPI_Recv

```
C:
int MPI_Recv(
  void* buf, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm, MPI_Status *status)

Parameters:
  OUT buf: initial address of receive buffer (choice)
  IN count: nb of elements in receive buffer (non-neg. int)
  IN datatype: datatype of receive buffer elements (handle)
  IN source: rank of source or MPI_ANY_SOURCE (integer)
  IN tag: message tag or MPI_ANY_TAG (integer)
  IN comm: communicator (handle)
```

```
  OUT status: status object (Status)

Python native:
recvbuf = Comm.recv(self, buf=None, int source=ANY_SOURCE,
    int tag=ANY_TAG, Status status=None)
Python numpy:
Comm.Recv(self, buf, int source=ANY_SOURCE, int tag=ANY_TAG,
    Status status=None)
```

Bibliography

[1] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[2] S. Adve and H.-J. Boehm, "Memory models: A case for rethinking parallel languages and hardware", *Communications of the ACM*, vol. 53, pp. 90–101, 2010.

[3] G. Amdahl, "The validity of the single processor approach to achieving large scale computing capabilities", in *Proceedings of the AFIPS Computing Conference*, vol. 30, 1967, pp. 483–483.

[4] Apache. (n.d.). Hadoop wiki, [Online]. Available: `http://wiki.apache.org/hadoop/FrontPage`.

[5] M. Bohr, "A 30 year retrospective on Dennard's MOSFET scaling paper", *Solid-State Circuits Newsletter*, vol. 12, no. 1, pp. 11–13, 2007.

[6] ——, "The new era of scaling in an soc world", *ISSCC*, pp. 23–28, 2008.

[7] J. G. Brookshear, *Computer science: an overview*. Addison-Wesley Publishing Company, 2008.

[8] A. W. Burks, H. H. Goldstine, and J. von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument", Tech. Rep., 1946.

[9] A. Chandrakasan, R. Mehra, M. Potkonjak, J. Rabaey, and R. Brodersen, "Optimizing power using transformations", *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, pp. 13–32, 1995.

[10] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, ser. Scientific Computation Series. MIT Press, 2008, vol. 10, ISBN: 9780262533027.

[11] L. Dalcin. (n.d.). MPI for Python, [Online]. Available: `https://mpi4py.bitbucket.io/`.

[12] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters.", in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[13] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions", *Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[14] E. W. Dijkstra, "Cooperating sequential processes", Tech. Rep., 1963. [Online]. Available: `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html`.

[15] ——, "Programming considered as a human activity", *Classics in software engineering*, pp. 1–9, 1979.

[16] V. Eijkhout, *Parallel Programming for Science and Engineering*. 2017. [Online]. Available: `http://pages.tacc.utexas.edu/~eijkhout/istc/istc.html`.

[17] V. Eijkhout, E. Chow, and R. van de Geijn, *Introduction to High Performance Scientific Computing*. 2016. [Online]. Available: `http://pages.tacc.utexas.edu/~eijkhout/istc/istc.html`.

[18] M. Flynn, "Some computer organizations and their effectiveness", *IEEE Trans. Comp.*, vol. 21, p. 948, 1972.

[19] A. Y. Grama, A. Gupta, and V. Kumar, "Isoefficiency: Measuring the scalability of parallel algorithms and architectures", *IEEE Parallel Distrib. Technol.*, vol. 1, no. 3, pp. 12–21, 1993.

[20] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using Advanced MPI: Modern Features of the Message-Passing Interface*. MIT Press, 2014.

[21] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, *MPI: The Complete Reference, Volume 2 - The MPI-2 Extensions*. MIT Press, 1998.

[22] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, 1999.

[23] W. Gropp, T. Sterling, and E. Lusk, *Beowulf Cluster Computing with Linux*, 2nd. MIT Press, 2003.

[24] D. Guttman, M. Arunachalam, V. Calina, and M. Kandemir, "Prefetch tuning optimizations", *High Performance Pearls*, vol. 2, J. Reinders and J. Jeffers, Eds., pp. 401–419, 2015.

[25] D. Heller, "A survey of paralel algorithms in numerical linear algebra", *SIAM Review*, vol. 20, pp. 740–777, 1978.

[26] J. L. Hennessy and D. A. Patterson, *Computer Architecture – A Quantitive Approach*, 3rd edition. Morgan Kaufman Publishers, 2003.

[27] M. Herlihy and N. Shavit, "The topological structure of asynchronous computability", *J. ACM*, vol. 46, no. 6, pp. 858–923, 1999.

[28] C. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985, ISBN: 978-0131532717.

[29] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, "Integrated pvm framework supports heterogeneous network computing", *Computers in Physics*, vol. 7, no. 2, 1993.

[30] L. Kale and S. Krishnan, "Charm++: Parallel programming with message-driven objects", in *Parallel Programming using C++*, MIT Press, 1996, pp. 175–213.

[31] M. Karbo, *PC architecture*. 2005. [Online]. Available: `http://www.karbosguide.com/books/pcarchitecture/chapter00.htm`.

[32] R. H. Landau, M. J. Páez, and C. C. Bordeianu, *A Survey of Computation Physics*. Princeton University Press, 2008.

[33] J. D. C. Little, "A proof of the queueing formula $L = \lambda W$", *Operations Research*, pp. 383–387, 1961.

[34] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor", *SIGOPS Oper. Syst. Rev.*, vol. 30, no. 5, pp. 2–11, 1996.

[35] S. Otto, J. Dongarra, S. Hess-Lederman, M. Snir, and D. Walker, *Message Passing Interface: The Complete Reference*. MIT Press, 1995.

[36] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference, Volume 1, The MPI-1 Core*, second edition. MIT Press, 1998.

[37] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI*. The MIT Press, 1994.

[38] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious", *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, 1995.