Parallel Computing

Dr. Juhász Zoltán

2014

A tananyag a TÁMOP-4.1.2.A/1-11/1-2011-0104 "A felsőfokú informatikai oktatás minőségének fejlesztése, modernizációja" c. projekt keretében a Pannon Egyetem és a Szegedi Tudományegyetem együttműködésében készült.

A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

1 PREFACE

Parallel computing is concerned with the execution of programs on computers having two or more processors. The main purpose of using multiple processors is to reduce the execution time of our programs. There has always been computational problems for which the computing power of a single processor is not sufficient. Traditionally, these problems have been the focus of parallel computing. There is a new, and much more practical reason for which parallel computing has become the new star of computing; and that is multi-core processors. We are in the multi-core era of computing; simply put, there are no computers with single-core processors any longer. The performance of processors is measured in the number of computing cores, not in clock frequency. A *sequential* program, written for one core will never be able to harness the performance of a multi-core processor. To maximally utilise these new processors, we need *parallel* programs that can be executed on different cores at the same time, in other words, *in parallel*.

In order for a program to be executable on a multiprocessor machine, it has to contain parts (subtasks or processes) that can be run on each processor in parallel. Our purpose in doing so is that by breaking the solution of a problem into sub-problems, we can decrease the overall execution time of the program. This concept is very similar, for instance, to the practice followed by the manufacturing industry, where a product is a result of components manufactured at the same time and then assembled in the most optimal way using several workers working in parallel.

The purpose of this book is to provide an introduction to this interesting topic and explore different aspects of parallel computation, such as multiprocessor computer architectures, parallel programming languages, parallel algorithms and applications. The amount of material that we could possibly cover in this book is enormous; therefore, the focus is only on introducing the most fundamental concepts, methods and programming techniques to students, which should prepare them for further, more specialised study of parallel computing subjects. It is always difficult to decide what topics to cover and what to leave out. Parallel computing is a much broader area than other fields of computer science; it affects each field of computing. Ideally, parallel computing should not be taught as a single course; rather, every topic of the computing curriculum should embrace and include parallel computing techniques and discuss them within the context of the field, for instance, present parallel architectures in Computer Architecture, parallel algorithms in Algorithms and Data Structure courses and so on. But until that happens, this book can help in guiding the reader through the key concepts.

The content of this book is strongly related to the structure and content of the Parallel Programming course regularly offered to BSc and MSc students at the Faculty of Information Technology at the University of Pannonia in Veszprem, Hungary. While the book is meant to be introductory, familiarity with the C and/or Java programming languages and knowledge of the fundamental concepts of computer architecture, operating systems, data structures and algorithms is expected. The chapters do not necessarily need to be read in the order they are presented in the book although, for readers not familiar with the topic, this might be the most suitable sequence.

While every care has been taken to create a correct manuscript, errors and omissions might be present. Any of these is the responsibility of the author but readers are more than welcome to send their feedback in order to improve the book.

Zoltan Juhasz

Table of Contents

2 INTRODUCTION

The fundamental reason for using parallel computing technology is to increase computing performance. There are always problems, applications that require more computational power that a single processor (processor core) can provide. Parallel computing is a practical answer to this problem, i.e., we want to solve the problem *now*. We cannot or do not want to wait for technological improvements (e.g. higher clock frequency) or new devices (optical or quantum computers); we need a solution in a practical timeframe. Thus, our only option is to use more of the readily available computing components.

But why exactly do we want ever increasing computing power? Why cannot we just stop and accept the currently available speed? What are the applications that require larger computational performance than what is currently available with sequential, single-processor/core machines. Here is a list of examples for such problems.

Scientific calculations – These include the traditional computational "Grand Challenge" applications, which are mainly simulations based on numerical calculations in the areas of physics, engineering, medical imaging, biological modelling, mathematics, weather prediction and so on. If, for instance, it takes 5 days to calculate the weather prediction for the next 2 days, there is not too much point in doing so. However, if we could decrease this time, say, to 10 hours, we could perhaps double or quadruple the accuracy of the prediction model.

Multimedia systems – The sheer amount of data in digitised images, video and sound files can be processed within reasonable time only with parallel computers. With sequential machines, even the simplest operation on a video sequence would take an unacceptably long time.

Database systems – Large databases create problems when large numbers of users try to perform complex queries. Search results can only be provided in reasonable time if several processors are working on different queries.

Operating systems – Current operating systems routinely run several tens of processes simultaneously. If these processes are spread onto several processors, instead of using only one, the performance and responsiveness of the computer will greatly increase.

Services on the Internet – Popular Internet applications and services (e.g. Google search, Amazon, Facebook, etc.) require server systems that can service millions of clients simultaneously in very little time. Parallel and distributed computing is the enabling technology for these systems.

For the past decades, the steady increase in the speed (clock frequency) of microprocessors guaranteed improved performance for the same program. Parallel computing was consequently a specialised area for those who wanted orders of magnitude larger computing power than available in the given or coming year. This has changed radically in the past few years. The absolute speed in the Universe is the speed of light. Even if we had working optical computers, we can never achieve execution speed greater than the speed of light. With currently available semiconductor technology, CMOS electronic components are used to build computers, consequently the speed limit is imposed by the propagation speed of electrons in silicon. Even if microprocessor manufacturers would achieve higher and higher operation speed in devices for many years to come, the speed of single-processor performance is bound by these physical limits. More fundamental engineering and manufacturing constraints, such as power consumption, heat dissipation, etc. brought an end to further increase in clock frequency. If we want more powerful computers in the future, we have two possible solutions.

- We invent and use fundamentally new computing technologies that can provide much greater computing speed and performance (e.g., optical, quantum or molecular computing).
- We try to use the existing technology in a more effective way, i.e. use several processors together and run parallel programs on them that are scalable to hundreds or thousands of cores.

Although there are encouraging experimental results with alternative computing technologies, currently the viable option is the second one, i.e. the use of more currently available processors. This can actually provide a very cost-effective way of creating powerful computers since the VLSI technology and the device speed do not necessarily have to be improved radically. The virtually unlimited increase in computing power hence is obtained by increasing the number of computing components in the system! This has been demonstrated in practice with the advent of multi-core processors and the many thousands of processors used in current supercomputers.

2.1 PARALLEL COMPUTING HARDWARE AND SOFTWARE

Just as for any sequential computing application, we need parallel hardware and software to create and execute a parallel program. The principal components of a parallel computer are a *set of processors* P_i, $i = 1,\ldots,p$, one or more *memory blocks*, and an *interconnection system* that connects the processors and memory blocks together in an appropriate way. Depending on whether processors are connected to memory blocks or to other processors, we differentiate between shared memory and distributed memory parallel computers, accordingly. Figure 2-1 illustrates a typical shared memory and distributed memory parallel computer architecture. There are several variations of both types of architecture and especially of interconnection systems. Chapter 5 will describe these parallel architectures in detail.

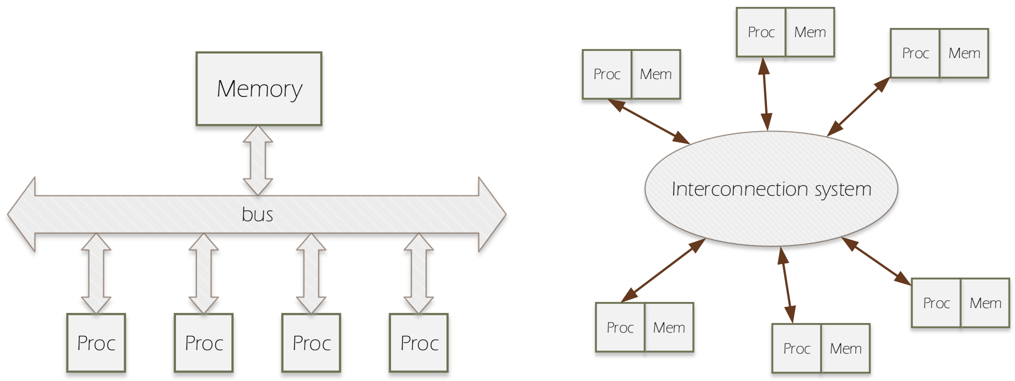


Figure 2-1 structure of a typical shared-memory and distributed memory parallel computer. The word Proc stands for processors while Mem for memory modules.

It is easy to see that if a problem is to be solved by a set of *processes* (program components executed in parallel), then processors will have to interact during the execution of the program. Most parallel programs require processes to exchange data. In shared memory computers, each processor can access the same global memory (address space), hence each can access variables that are used by others. As a consequence, processes running on shared-memory computers usually communicate through memory via shared variables. On the other hand, if memory is local to each processor, as in distributed-memory parallel computers, one processor cannot access data at a memory location of another processor. Therefore, they usually send messages to one other requesting or sending data from or to the given process. The underlying hardware architecture has profound effects on the choice of programming languages and on the performance of the applications, as we will see later in the book.

Once we have a parallel computer, we then need a parallel solution to the problem in question and a parallel programming language to express the solution in a way that is understandable for the machine. The aim of *parallel programming* is to express the solution of a problem in terms of smaller tasks and execute these subtasks in parallel to solve the original problem in shorter time. We are interested in firstly, how a given problem can be decomposed into a collection of parallel tasks and secondly, how we could express the parallel solution with a programming language. The first issue is about parallel program design, while the latter is on how to write a correct parallel program.

The key concept in a parallel program is the *process*. A process is a piece of sequential program code with its own data and state. A parallel program, hence, is a collection of processes. In parallel programs therefore, we must be able to manage processes (e.g., creating, stopping, etc—these features are not required in traditional sequential languages), and need language constructs that help in exchanging data among processes (communication) and synchronizing their actions.

The idea to divide a problem into smaller sub-problems, and then solve them in parallel at the same time sounds very simple, but unfortunately, as we will see throughout this book, things are much more complicated in practice. There are no clear-cut solutions as to how to create efficient parallel programs. But there are few general guidelines, approaches that can be used to help us in this task. The two main approaches to problem decomposition are based on partitioning either the input data set or the algorithm itself. Details of these methods are discussed in Chapter 3.

We have said earlier that there are two principal ways of exchanging data among processors: either by using shared memory variables or by message passing. To the analogy of the hardware models we will differentiate between programming languages based on *shared variables* and those using *explicit message passing*. Chapters 6 and 7 give an overview of the shared variable and message passing programming fundamentals along with the introduction of the most widely used parallel programming languages. Chapter 8 continues with examples of key parallel algorithms and examples of how they can implemented using different parallel languages and environments.

A parallel computing system that provides exceptional computing performance is almost a piece of art; a very complex masterpiece of engineering that is the result of careful combination of semiconductor, computer architecture, software engineering, algorithm and application design expertise. As it builds on so many types of knowledge, concepts of parallel computing should be present in computer science/engineering course and should not be taught in isolation.

This book intends to introduce the reader to the main areas of parallel computing technology. **Part 1** provides a general introduction to the parallel computing landscape and show the main paradigms that we can use to turn sequential problems parallel. **Part 2** concentrates on parallel computer architecture and examines the ways to construct parallel computers from processor and memory components. Both shared and distributed memory architectures are covered. **Part 3** focuses on parallel programming language issues. After an introduction to the problems of concurrency and parallelism in a program, it explains thread-based programming in Java, and describes the use of the OpenMP and Message Passing Interface standards. **Part 4** describes some of the most common parallel algorithms and provides implementation examples.

3 Parallel programming fundamentals

3.1 Computational performance

Parallel computing is about computational performance. Without properly measuring the performance we cannot reason about our algorithms or compare different computers with one another. Since the main use of parallel computers is for numerical computations in the computational science community, the key metric of computing performance is the number of *floating-point* numerical *operations* the computer can perform in *a second*. From the abbreviation of this long term came the unit *flops* or *flop/s*. Since processors can execute a very large number of operations in a second, we normally measure them in larger units, such as Mflops (mega 10^6), Gflops (giga 10^9), Tflops (tera 10^{12}), Pflops (peta 10^{15}) and Eflops (exa 10^{18}). In comparison, a typical desktop processor core has a performance typically in the range of 1-15 Gflops running at 100% CPU load.

3.2 Supercomputing and parallel computing

Since the invention of digital computers, engineers were engaged to design and build a faster computer than the existing ones. The fastest computers in the world are historically called *supercomputers*. Computing on these computers is called *supercomputing* and the companies making these computers and the associated software packages together form the *supercomputing industry*. Supercomputing really started in the 1960s, when solid state semiconductors allowed sufficient size reduction of the required components. Seymour Cray founded Cray Research who created the Cray-1 (1976), shown in Figure 3-1, that achieved approximately 80-130 Mflops performance [1]. Several new models followed, like the Cray-2 (1985, 300 Mflops), the Cray X-MP and Cray Y-MP (1988, up to 15 Gflops), dominating the supercomputer industry till the 1990's. The cost of these computers was in the range of million US dollars. The most important thing to note is that from the Cray-2 model onwards these computers used multiple processors to achieve supercomputer performance. Parallel computing and supercomputing technology development hence went hand-in-hand.

In the 1990s, the number and the speed of processors increased higher and higher and soon, supercomputers achieved the 100 Gflops performance range. Japanese and US manufacturers competed for the fastest computer title. Notable supercomputers from this era are the Fujitsu

Numerical Wind Tunnel (1994, 166 processors, 124 Gflops), the Hitachi SR2201 (1996, 2048 processors, 600 Gflops), the Intel Paragon (1994, 3680 processors, 143 Gflops) and the Connection Machine CM-2 (1987, 65 536 1-bit processors, 6 Gflops) and CM-5 (1993, 1024 processors, 131 Gflops).

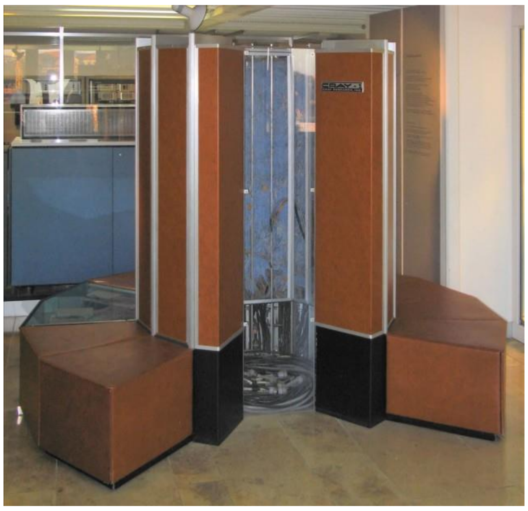


Figure 3-1 The Cray-1 computer (Cray Research) *[2]*

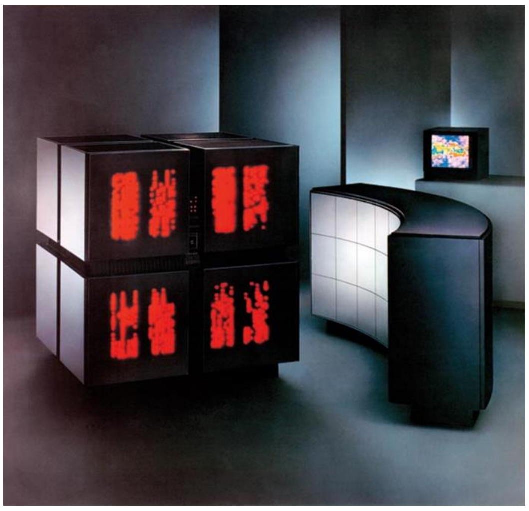


Figure 3-2 Connection Machine 2 (Thinking Machine Corp.)[1]

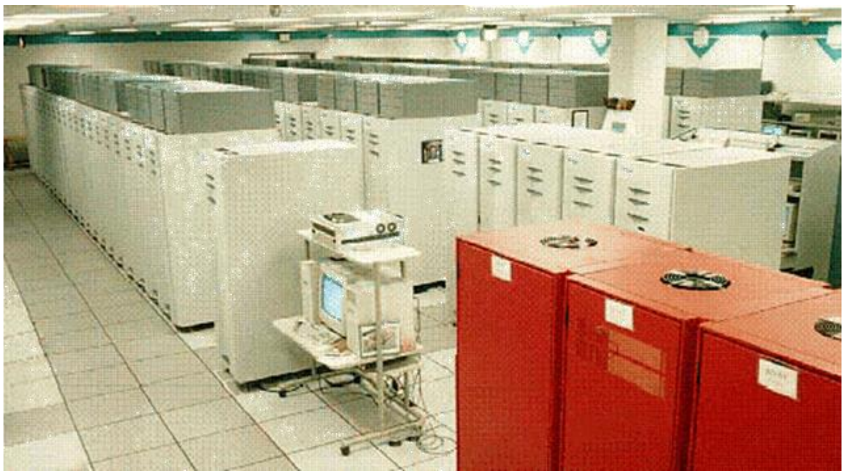


Figure 3-3 ASCI Red, the first teraflops computer[2]

[1] Image source: http://www3.sympatico.ca/n.rieck/images/TM-CM2-0109-SUPERCOMP_x600.jpg

[2] Image source: http://www.computermuseum.li/Testpage/ASCII-RED-Supercomputer.gif

Having created many computers in the hundred Gflops range, the supercomputing industry was after its next challenge: to achieve teraflop computing performance. This was achieved by the US research and development program ASCI (Accelerated Strategic Computing Initiative) [3] whose aim was – among others – to build several teraflops supercomputer from commercial, off-the-shelf components. This decision started an important development path that resulted in computers that could be more easily upgraded and supported the porting of parallel software packages better. The result of this effort was unquestionable since the first computer breaking the Tflops barrier in 1997 was the ASCI Red supercomputer (2.4 Tflops). Not long after a successful series of Tflops-range ASCI computers (ASCI Red, White and Blue), Japan created the Earth Simulator supercomputer that achieved 32 Tflops performance and became the first teraflops computer built outside the US.

By 2008, the supercomputing industry reached the petaflop performance range. The IBM RoadRunner computer was the first to break the petaflop barrier achieving 1.026 Pflops sustained performance in 2008. Development has not stopped since, and at the end of 2013, the world's fastest computer – the Chinese Tianhe-2 – works at over 33 petaflops speed (shown in Figure 3-5).

Figure 3-4 The first petaflops supercomputer – RoadRunner, IBM Blue Gene/P[3]

[3] Image source: http://upload.wikimedia.org/wikipedia/commons/d/d3/IBM_Blue_Gene_P_supercomputer.jpg

The development of the most powerful supercomputers in the world can be followed by studying the TOP500 list that measures the performance of the most powerful computers twice each year (http://www.top500.org). Very interesting trends and developments can be discovered from the data sets of this archive. At the time of writing, the five most powerful supercomputers in the world are the followings. The readers are advised to consult the TOP500 list for further details.

Table 3-1 The top five supercomputer in the world (as of November 2013).

Rank	Name/model	Country	Number of cores	Performance [petaflops]	Power consumption [megawatt]
1	Tianhe-2	China	3 120 000	33.8	17.8
2	Titan - Cray XK7	USA	560 640	17.6	8.2
3	Sequoia BlueGene/Q	USA	1 572 864	17.1	7.9
4	K Computer	Japan	705 024	11.3	12.6
5	Mira BlueGene/Q	USA	786 432	8.6	3.9

Figure 3-5 Tianhe-2 supercomputer, China[4]

[4] Image source: http://www.engineering.com/Portals/0/BlogFiles/DesignerEdge/1113/Tianhe-2.jpg

3.3 PARALLEL COMPUTERS FOR THE MASSES

Needless to say, these supercomputers are very expensive, their cost can reach 100 million USD per system and use megawatts of energy. Very few countries in the world can afford the installation and operation of such giants. If parallel computing was only about supercomputers, not many of us could ever run a parallel program. Fortunately, there are more economical alternatives that – for an individual user – may provide just as good performance. Network connected desktop computers, also known as computing clusters, can easily achieve teraflops range performance at no additional cost. This form of computing is especially popular at universities where existing laboratory computers can be efficiently utilised for solving computationally intensive problems. Assuming 4-core computers, a lab with 50 computers (200 cores) can achieve around 2 teraflops performance. Another alternative is to use graphics cards either as single units or as components in clusters. Top graphics card today can achieve 3-4 teraflops computing performance at a fraction of the cost of a PC. Using about 30 such cards together, one can achieve over 100 teraflops. For students entering the world of parallel computations, common dual or quad-core notebook computers are perfectly adequate to explore techniques and study parallel algorithms and program development.

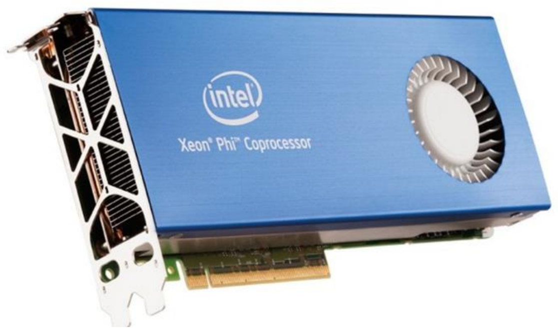


Figure 3-6 Xeon Phi, Intel's 60-core, 1Tflops HPC co-processor [4]

Figure 3-7 NVIDA Tesla K40 GPU accelerator, 2880 cores, 4.29 Tflops [5]

3.4 SOLVING PROBLEMS IN PARALLEL

Having looked at parallel computers of different sizes, one starts to wonder whether there is a method that we can follow to make an existing sequential program suitable to run on a multiprocessor computer. Is there perhaps some 'magic' that can transform a program into a parallel form? The answer is that automatic parallelisation of a sequential program only works on special problems and in limited situations. Normally, the programmer has to design the parallel structure and operation of the program. Unfortunately, there is no universal recipe for creating parallel programs, or in other words, parallelising sequential programs. There are tools that can help but, in the end, it is the program designer who must make the key design decisions.

Students are normally trained to solve computer programming problems in a sequential fashion. We are taught that a computer executes instructions one after the other in strict sequential order. If we want to change the order, we use branching instructions. These fundamental concepts – sequence, branching and loops – are also used in parallel programs but we need to liberate ourselves by allowing several tasks to proceed at the same time. If we think of analogies, we will discover soon that life around us is in fact parallel; e.g., we hardly ever build a house or manufacture a product using one single worker. To save time, we would use many workers, often having different skills, in a coordinated way. A good work manager is also needed who ensures that workers will have access to resources and can perform their tasks in the most optimal way.

At the very highest level, our problem is reduced to finding work units that can be assigned to processors, ideally without any dependency from others. The art of parallel computing is to discover the most efficient way of splitting a solution to smaller sub-problems and make it suitable for the target computer that can execute it with maximum efficiency. We will see later in this book that creating parallel programs is not always trivial. It requires time and energy to develop an efficient parallel program. One has to be careful in judging whether or not a parallel implementation is beneficial. For instance, there is little use of spending weeks on parallelising a program that will only be run once and the execution time of the program is, say, few minutes. We are better off waiting few minutes instead of working for weeks. Remember that parallel computing is about speed. If the outcome of our effort cannot justify the parallel programming effort, do not do it. If, however, the potential benefits are great indeed, try to find every trick to make the program run as fast as possible!

The starting point for solving any problem in parallel is to find sub-problems that can be executed by different processors at the same time. This method is called *partitioning*. By finding smaller and smaller tasks, we potentially increase the level of parallelism in the solution. Depending on how many sub-problems we end up with, we differentiate among *low*, *medium* and *high-granularity* parallelism. An algorithm has low granularity if the tasks are built up from many larger functions. A medium granularity problem can have tasks executing instructions at the function level. In high-granularity parallel programs tasks are instruction level entities.

Besides partitioning, another key concept in parallel computing is *efficiency*. It is no use to have many parallel tasks if they cannot run on the system efficiently. If tasks are delayed for any reason, the time wasted – also known as overheads – will reduce the overall performance of the system. In the worst case, tasks can even execute sequentially if resource or other dependencies dictate that way. Overheads can arise from communication delays, synchronisation with other tasks, not having enough computing resources or work to execute. We will investigate these issues later in more detail. In the rest of this section, we will look at the most common forms of parallelism that we can effectively use in our parallel programs. Remember, there are no 'recipes' to perfect and automatic parallelisation, each problem requires a fresh look, but based on experience gathered over the past two decades, we can identify distinct approaches to or 'styles' of parallel executions. These are based on *what* we make in the program parallel and *how* the parallel processes are then executed.

3.4.1 Data or geometric parallelism

There are many problems where the solution requires repetitive computations on varying data. Typical examples are transformations of one, two or three-dimensional matrices, processing pixels in images, simulating physical properties of objects. The core of the computation is the same for each data items, hence normally a loop is used to iterate over the data set. A very simple example is the calculation of the square of the first *N* natural numbers. The sequential code would be the following:

```
for (int i=1; i<=N; i++)
{
	result[i] = i * i;
}
```

A single core will finish the operation in *N* steps. If each step takes *T* time to complete, the entire operation will execute in *N·T* time. If we visualise the data set, we see a one-dimensional array, a vector that the program fills up with the value i^2, where *i* = *1…N*. Notice that the iterations of the loop do not depend on each other, i.e. the calculation of each iteration can be done independently of the others. Let us divide now this set into *k* non-overlapping equal parts, where *p* = 2…*N*, and we end up with parts having *N/p* items. Now, we assign each part to a processor and assume they execute the same code on the smaller data sets at the same time. Figure 3-8 illustrates this process for the *p* = 2 case, i.e. using 2 processors.

The original loop will be split into two loops, one for each process. Process 1 will execute the loop for the first N/2 items, Process 2 will perform the second N/2 items. The code for the two processes will be as follows.

Process 1:

```
for (int i=1; i<=N/2; i++)
{
       result[i] = i * i;
}
```

Process 2:

```
for (int i=N/2+1; i<=N; i++)
{
       result[i] = i * i;
}
```

In the ideal case, the program will finish in half of the original execution time. As we increase the value of *p*, more and more processors are used and the execution time is reduced. Using *N* processors, the program will finish in 1 time unit.

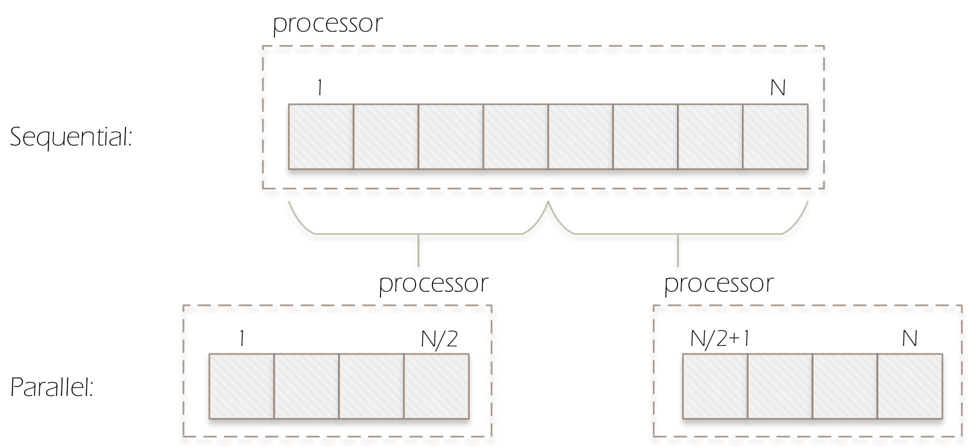


Figure 3-8 Partitioning a data set for data-parallel execution.

The same concept can be used for higher-dimensional data structures. As an aside, we note that the granularity varies with the data size – processor number ratio. For a given data size *N*, if we have 2 processors, on the other hand, if the number of processors are close to the data size, the granularity is high, the processors will work on very few data items.

Let us continue with two examples of a two-dimensional and three-dimensional computational problem.

Image processing

Imaging application have to manipulate pixels in large digital images. Even a simplest operation, for example, increasing the intensity or contrast, require lengthy computations as each pixel must be processed in the image. Figure 3-9 illustrates the sequential and parallel approach to this typical processing strategy. In the sequential case, the image is represented as matrix with N rows and columns (for simplicity, we assume square images). A double nested **for** loop is used to visit each pixel and apply the required operator. With today's technology, the number of pixels in the images range from 10^6 (mega) to 10^9 (giga) pixels.

Since the pixel operations are independent form each other, they can be executed in a data-parallel fashion. One possible positioning of the image data is shown in Figure 3-9. The image is split into smaller square tiles of size $M \times M$, where $M = N / \sqrt{p}$ and p is the number of processors. Using 16 processors, as in the example, each processor will work on an $N / 4 \times N / 4$ sub-image and the execution time of the entire operation is reduced to the 1/16th of the original.

Data parallelism is a very natural way of dividing a large computational problem into simultaneously executable parts. The main advantage of this approach is that the parallel program can be easily adjusted to the number of available processors. If there is no data dependency among the parallel tasks, near ideal performance can be achieved. Luckily, a very large part of the computational problems (especially scientific – numerical – calculations) fall into the data parallel category.

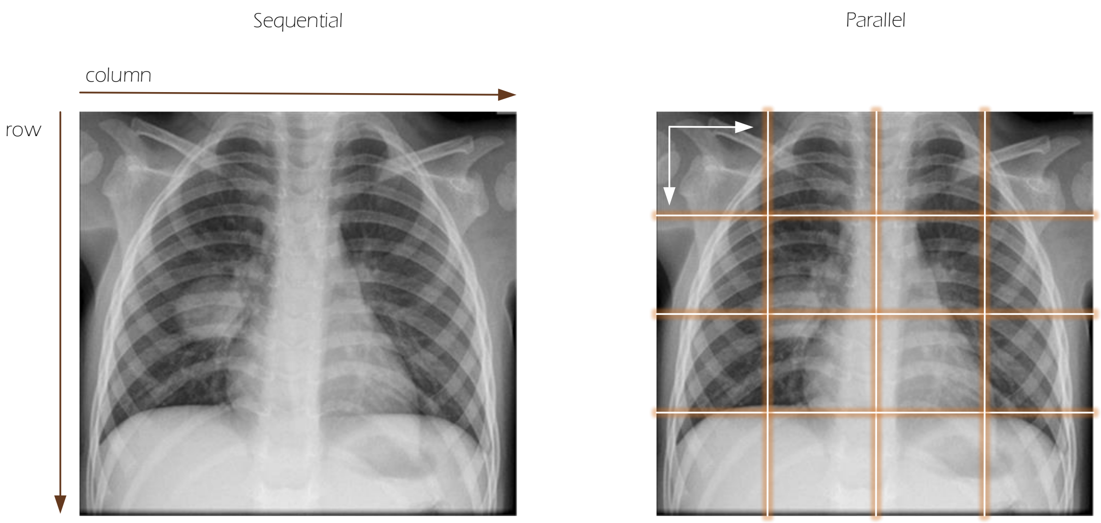


Figure 3-9 Data-parallel partitioning of an X-ray photograph for pixel manipulation.

3.4.2 Control or algorithmic parallelism

There are problems whose solution requires complex computational steps and the execution time is determined by the operations, not the size of the data. In fact, in many cases, the data set is constant or small. Our only hope to speed up such programs is to execute different instruction sequences (partial calculations) in parallel. This method is called control or algorithmic parallelism. Ideally, our problem can be divided into partially or fully independent sequences of statements (tasks).

We illustrate the algorithmic parallelism in a simple example. Assume that we need to compute the roots of a quadratic equation. For the sake of simplicity, we assume that execution time of the floating point operations is so great that we must create a parallel implementation for the calculation. The roots are given by the well-known equation:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

A sequential program would take the three input parameters a, b, and c, and evaluate the terms of the above equation in succession. The calculation of each root will require 9 floating point operations – assuming that the square root is one operation.

The input data set is always the three parameters, and the calculation is not repeated for a, b, and c, as in the data-parallel case. We need to find parallelism in the evaluation of the expression. Figure 3-10 depicts the execution graph of computing the root. The input variables are represented by circles, and the different floating point operations are shown in rounded rectangles. Each rectangle transforms the data it receives at the top and sends it forward at the bottom. If two inputs are given, the operation is executed from left to right inputs, i.e. left input = *x*, right input = *y*. This representation is also known as the dataflow graph as it clearly shows how data flows through the computing network as it is transformed. The result is produced at the last nodes of the graph at the bottom.

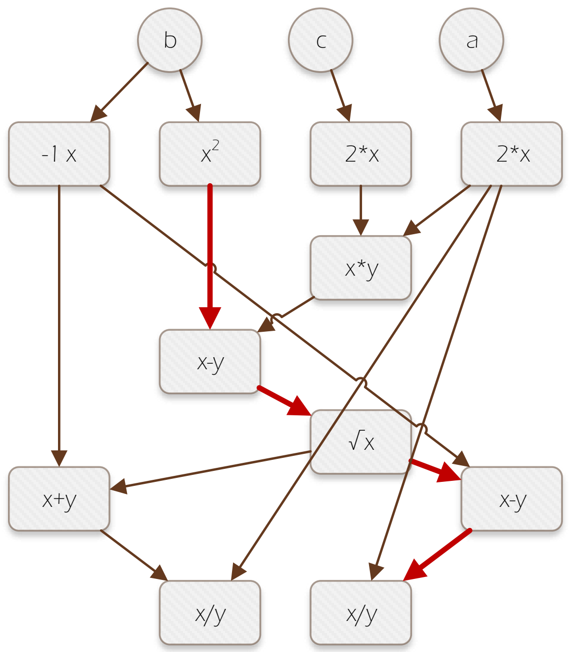


Figure 3-10 Execution graph of the quadratic equation root calculation. The red arrows indicate one critical path of the execution.

The question now is whether or not the different operations can be executed in parallel and if so, what is the level of parallelism in the system. The graph shows dependencies (on is highlighted in red) that must be executed serially, one after the other but also displays opportunities for parallel execution. By slightly rearranging the graph, we can create a so-called space-time diagram (Figure 3-11), commonly used in parallel computing which displays how the tasks are executed by each processor and in which order. Since the maximum number of independent processes are 4, the graph is mapped onto 4 processors. The diagram also indicates the time required for the execution of the calculation. Assuming that each operation takes 1 time unit, adding the rows from top to bottom produces the overall parallel execution time. In this case, 6 units. Since the original serial calculation required 9 operation, we are less than optimal. With four processors, we would like to see an execution time of 9/4 steps. We can see, that the use of the processors is less than optimal. None of the processors are busy constantly; processor p_4, for instance only performs one operation. By merging the tasks of processors p_1 and p_2, we can reduce the number of processor to 3 without increasing the execution time. While this step did not reduce execution time, it definitely made the overall system more efficient.

Even this simple example can show that the amount of parallelism in algorithmic parallelism is limited. It is hard to find problems and parallel solutions in which a large number of processes can execute in parallel. Moreover, it is quite difficult to find the independent parts in an execution graph, and partitioning an algorithm into *k* tasks does not mean we can execute all *k* tasks at the same time. There might be data dependencies that will require sequential execution of certain tasks! If we want large increase in computing power, we normally try to use the data parallel paradigm.

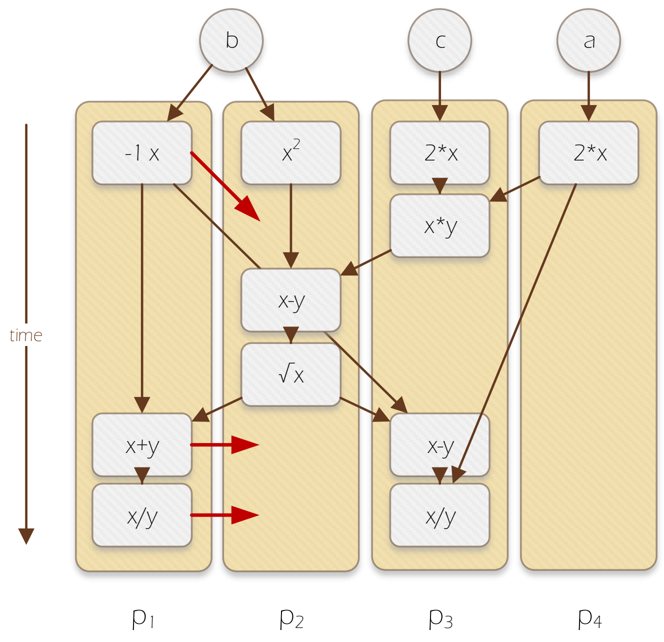


Figure 3-11 Space-time diagram of the execution of the graph in Figure 3-10. Red arrows indicate how tasks of p_1 can be moved onto p_2 without increasing execution time.

3.4.2.1 *Pipelining*

Pipelining is a special form of control parallelism - the output (the result) of one process becomes the input of the next one. The idea of pipelining is not new, it has been widely used in sequential processors and in vector supercomputers. Although the pipeline structure (shown in Figure 3-12) resembles sequential execution, it is in fact an efficient parallelisation strategy when data must be processed in successive stages to produce the final result.

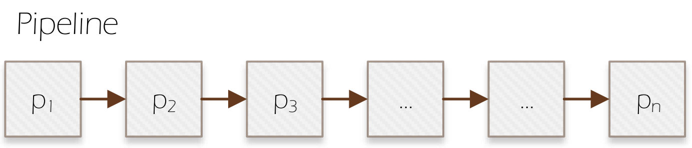


Figure 3-12 *Processing in a pipeline. Each process sends its output to the next one which will use it as an input data to work on.*

To understand how the pipeline can reduce execution time, let us look at the operation example shown in Figure 3-13. We assume that our data takes 5 sequential stages to be processed. These stages are assigned to processors in a five-processor pipeline. Data is then pushed into the pipeline at processor p_1 which will perform the calculation steps of the first stage of the processing steps then pass the result on to processor p_2. Processor p_2 then will do its share of the calculation and pass the result on to p_3 and each consecutive processor will work similarly. As we can see in Figure 3-13, in each time step, a work package travels to the next stage of the computation. See, how work package 1 travels from the start of the pipeline to the end in 5 time steps. When we start the calculation, the first work package is in processor p_1. All other processors are *idle*. When p_2 starts its computation, p_1 is already working on the second work package. At each time step one more processor becomes active in the pipeline.

Once the first package reaches the last stage of the pipeline, in this case p_5, the pipeline is said to be *filled*. From this moment, the pipeline will produce one result in every time step (see t_6 and t_7), even though the entire processing time for the input data takes 5 time steps. Therefore, the execution time is reduced from 5 to 1 units which is a factor of 5 performance gain over the sequential execution. More formally, in situations when the time of filling the pipeline is negligible to the overall execution time, the execution time becomes $T = T_{\text{sequential}} / P$, where P is the length of the pipeline. This explains how the pipeline can speed up calculations of transformation sequences, and show that the longer the pipeline becomes, the more we can reduce the execution time.

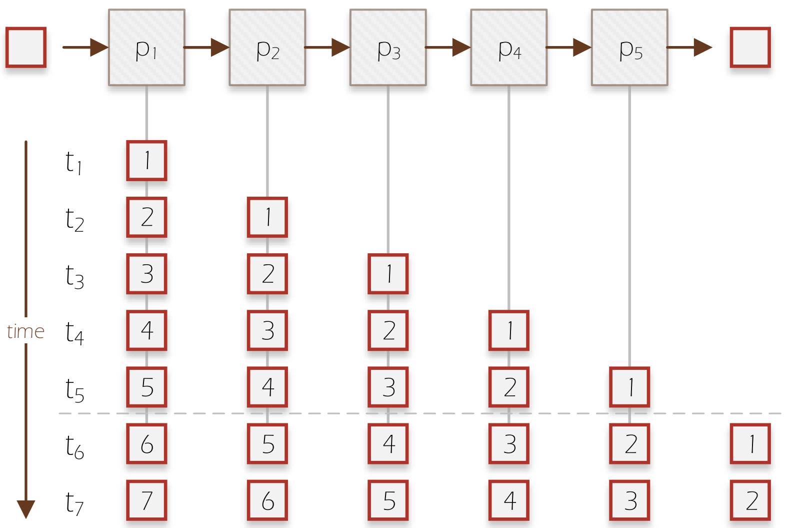


Figure 3-13 *The flow of work units in a pipeline.*

The above example used an idealised scenario in which each stage takes an equal amount of time to complete. This type of pipeline is called a *balanced* pipeline. Let us examine what happens if different stages of calculations take different amount of time in the pipeline. This is illustrated in Figure 3-14, where the execution time of processor p_4 is twice the time of the others. We can see that from the moment p_4 starts working, the execution time of each stage is determined by the execution time on p_4. This is because the processors work in a lock-step fashion. No matter how fast one processor completes its work, it will not be able to send or receive its next work package while p_4 is busy, hence, eventually p_4 will control the execution. Consequently the speedup in a general pipeline is given as the ratio of the sequential execution time (total work unit execution time) to the pipelined execution time (time of producing one result with a full pipeline):

$$S = \frac{\sum_{i=1}^{N} T_{p_i}}{\max\limits_{1<i<N} T_{p_i}}$$

This shows the weakness of pipelines: in order to gain large performance increase we need long pipelines and nearly identical processing times for each stage (processor). It is very difficult to find problems where the pipeline can be long $p > 10$ and also very hard to ensure that all processes will have equal execution times.

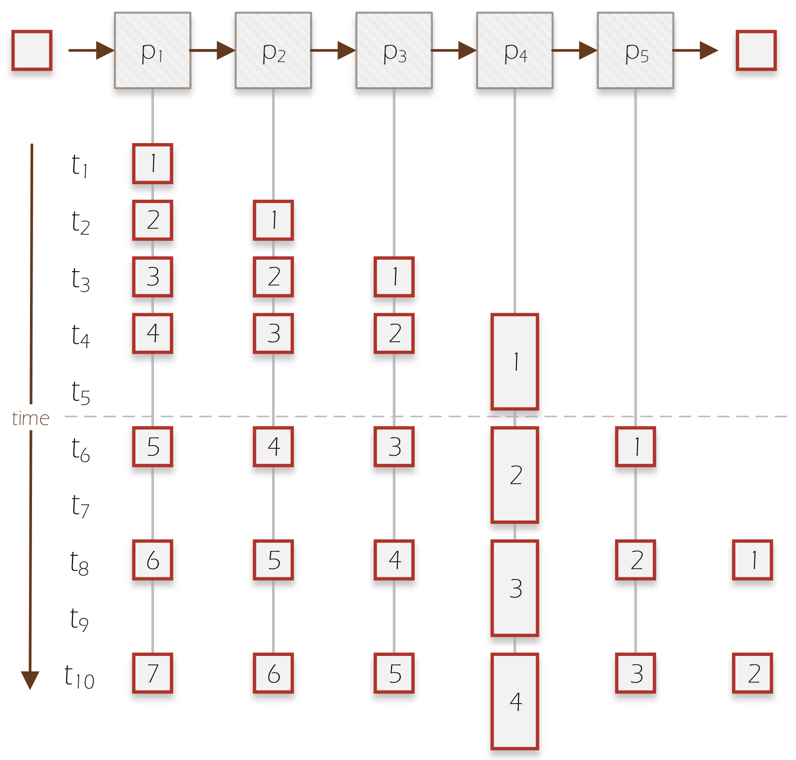


Figure 3-14 *Imbalanced pipeline in operation.*

3.4.3 Farming parallelism

In our discussion so far, we were looking at how to split the data or the algorithm of a problem into smaller parts. We implicitly assumed that the execution time was known before we partitioned the problem and the time of each subtask is always constant.

There are many examples for which this assumption is not valid. Either we do not know the execution time when we split the problems, or the time depends on runtime parameters, i.e., the computational cost of the subtasks changes dynamically. In these cases, both data and algorithmic-parallel programs can result in an inefficient use of the processors if some processors receive more work than the others; the less loaded processors will spend time waiting for others instead of doing useful computation. This is known as the computational *load imbalance* problem, which we will later discuss in more detail.

Farming parallelism, also known as Master/Slave or Master/Worker parallelism is a dynamic work allocation strategy that ensures that processors are uniformly loaded with work even in a dynamic environment. This strategy can be used equally well for data or control-parallel programs. We illustrate the operation of the farming parallelism strategy with a well-known computational problem, the calculation of the Mandelbrot Set. Figure 3-15 shows the visual representation of the Mandelbrot Set. This is a famous example of fractal geometry invented by Benoit Mandelbrot. The picture shows the results of the membership function of the following recursive function on the complex plain:

$$z_n = z_{n-1}^2 + c$$

where $n = 0, \dots N$, c is a complex number in the plane to be tested for membership and $z_0 = 0$. A point c is in the set only if $|z_n| \leq 2$ for all $n \geq 0$. The point of each point c is relative to the value of n where the condition $|z_n| \leq 2$ fails. Points that are in the set are coloured black.

Since the same program (membership test) part is executed for each point (pixel) of the plane, this is a potential data-parallel problem and could be divided into subparts as shown in Figure 3-15 ($p = 16$). The trouble with this approach is that the rectangular areas assigned to each processor will take varying time to execute and even worse, we do not know who long each part will run. This is because the membership test includes a conditional expression that can only be evaluated at runtime.

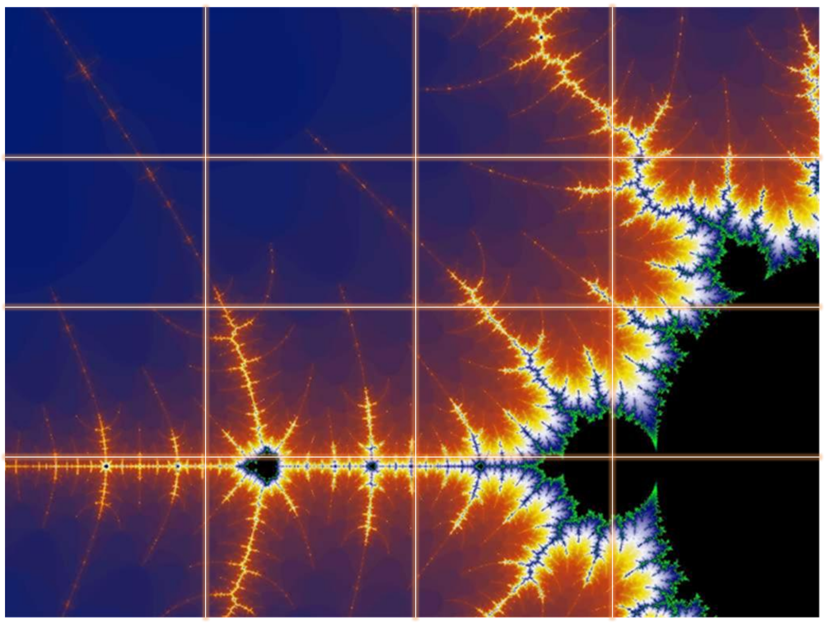


Figure 3-15 Visual representation of the Mandelbrot Set along with one possible work partitioning for 16 processors.

The farming parallelism approach uses a master controller acting as a work dispatcher to 'farm out' tasks to worker or slave processors. The operation is simple. Slaves only perform work. When a slave finished with the work, it returns the result to the master which, in turn, dispatches the next task to the slave. This cyclic pattern is repeated until all tasks are completed.

It is easy to see that if we use the exact same 16 work units and *p* = 16 processors as in Figure 3-16, the situation would remain the same. Each processor would execute for some (unknown) amount of time and when the last one finished, the image is complete. The central idea in farming parallelism is that the problem is divided in such a way that the number of computational units (work packages) is much larger than the number of processors. The many small work units statistically even out imbalance of the processors; those receiving tasks taking longer to compute will execute fewer tasks, the ones with short tasks will execute more. It has been shown that with this approach it is possible to keep processors busy and achieve good load balance and high performance.

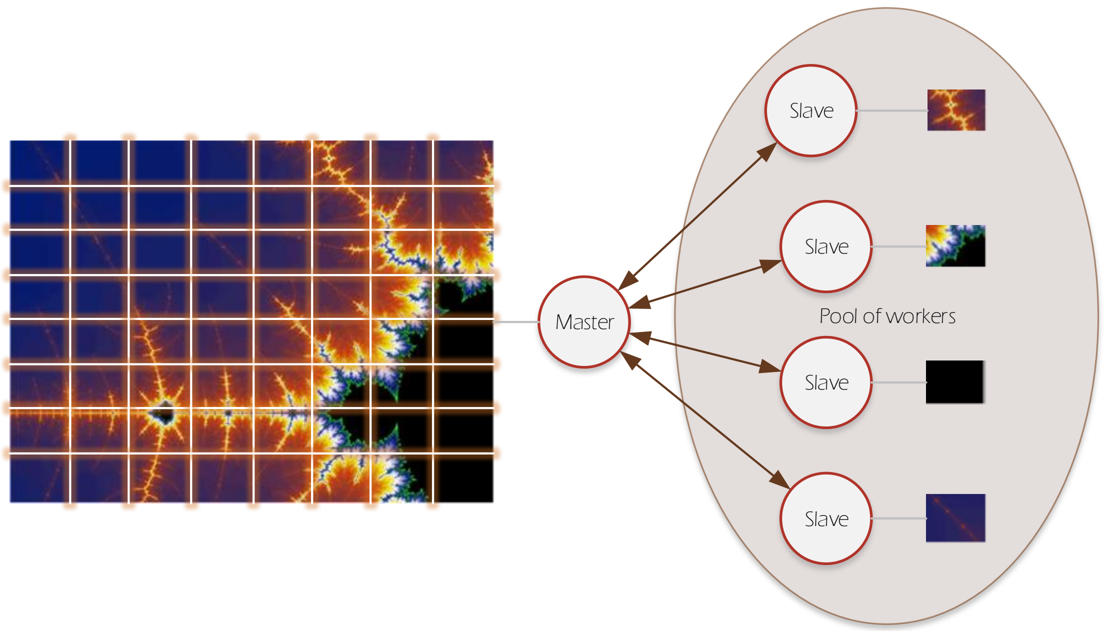


Figure 3-16 Using the farming parallelism approach for the Mandelbrot Set calculation problem.

3.5 Parallel Execution Performance – the Basics

In this section we look at the most fundamental metrics that can help measure the performance of a parallel system and the level of success in creating an efficient parallel program. We discussed ways of partitioning a problem into subtasks for parallel execution but we need objective metrics that inform us about the performance of the solution.

The execution time T_s of the sequential program we will parallelise will serve as a baseline to which we will compare the parallel system. Let T_p denote the execution time of the parallel program. Ideally, the parallel time is the sequential time divided by the number of processors, *p*, thus

$$T_p = \frac{T_s}{p},$$

which is the inverse (or reciprocal) function of *p*, so by doubling the number of processors (*p* = 1,2,4,8…64,…), the execution time will be halved as shown in Figure 3-17. When measured, however, we often get a sub-optimal value.

We define the ratio of the sequential to the parallel execution time as the *speedup* of the parallel system. The speedup, S_p, is a metric that indicates the execution speed gain of the parallel program over the sequential one and is defined formally as

$$S_p = \frac{T_s}{T_p}.$$

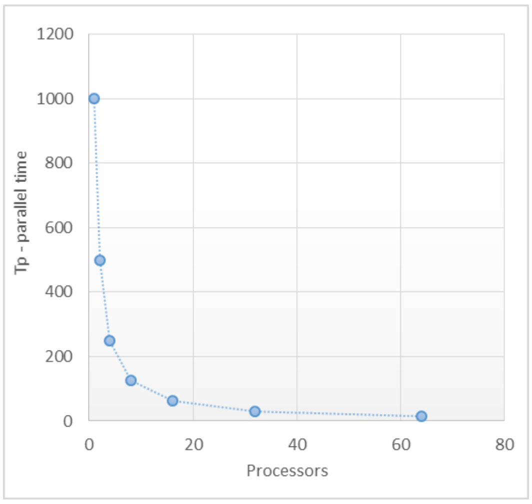
By substituting the ideal parallel time, $T_p = T_s/p$, into the speedup formula we see that in the ideal case

$$S_p = \frac{T_s}{\frac{T_s}{p}} = p,$$

that is, a speedup is equal to the number of processors. The graphical representation of the ideal speedup is shown in Figure 3-18. Theoretically, speedup should be derived using the sequential time T_s measured on the fastest processor running the most efficient sequential algorithm; this is the *absolute speedup*, showing how much faster our program is when compared to the fastest possible sequential execution. In practice, obtaining this best sequential execution time is almost impossible. For this reason, we typically use a more relaxed form of the speedup which is called the *relative speedup*, which is derived from the execution time of the sequential algorithm forming the basis of the parallel implementation on one processor of the parallel computer. This relaxed sequential execution time is denoted by T_1, and the relative speedup is defined as

$$S_p = \frac{T_1}{T_p},$$

where T_p is now meant as $T_p = T_1/p$.



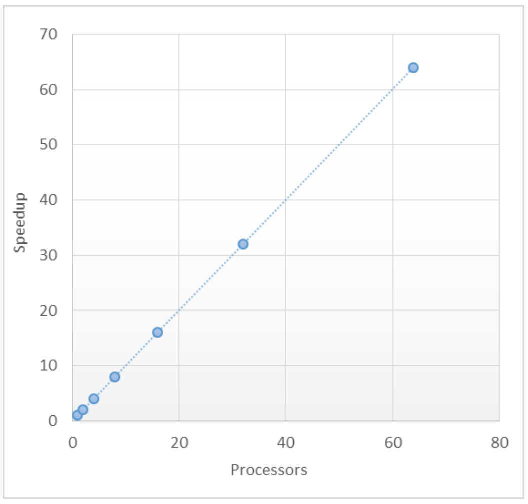


Figure 3-17 Parallel execution time

Figure 3-18 Ideal speedup curve.

Our last metric is *efficiency*, E, which describes how efficiently the parallel system achieves the measured speedup. Efficiency is defined as the ratio of speedup to the number of processors used in the system,

$$E = \frac{S_p}{p}.$$

Clearly, a speedup of 12 achieved on a 12-processor system is a far more efficient execution than the same achieved on a 64-processor one. From the definition of speedup we can see that the ideal efficiency is $E = 1$, that is, 100%.

3.5.1 Suboptimal performance

In real applications, achieving ideal performance is nearly impossible. There are various overheads owing to hardware, algorithm and execution properties that result in wasted time during execution (e.g. copying data to processors, sending work packages in a farming parallel program, synchronising processes). The total overheads have an important effect on performance.

Gene Amdahl was among the first researchers to warn against over optimism in parallel computing. In his paper [6] he states – what is now known as *Amdahl's Law* – that there will always be a portion of the program that cannot be made parallel and therefore will put a limit to the achievable speedup. Let α denote the part of the program that cannot be parallelised. Then, the parallel execution time can be expressed as follows:

$$T_p = \alpha T_1 + \frac{(1-\alpha)T_1}{p}$$

The first term of the expression is the time of the remaining sequential part while the second term is the ideal execution time of the fully parallel part. When we express the speedup using this expression and take its limit we get

$$\lim_{p\to\infty} S_p = \frac{T_1}{\alpha T_1 + \frac{(1-\alpha)T_1}{p}} = \frac{p}{\alpha p + (1-\alpha)}$$
$$= \frac{p}{1+(p-1)\alpha} = \frac{1}{\frac{1}{p} + \left(1 - \frac{1}{p}\right)\alpha} = \frac{1}{\alpha}$$

This limit effectively means that the speedup is bound and is independent of the number of processors. Put it simply, if 10% of the program cannot be made parallel, the maximum speedup we can achieve is $S_p = 10$ regardless the number of processors used!

This somewhat contradicts the fact that supercomputers are built with hundred thousands of processors. Surely, people would not spend millions of dollars without good reason. While Amdahl's Law is certainly valid in general, as John Gustafson pointed out [7], for most engineering and scientific problems, the sequential portion α becomes a function of the problem size n, that is, $\alpha(n)$, which normally becomes smaller as n increases:

$$\lim_{n\to\infty} \alpha(n) = 0$$

Taking the limit of the speedup using $\alpha(n)$ in terms of problem size n results in

$$\lim_{n\to\infty} S_p = \frac{p}{1+(p-1)\alpha(n)} = p$$

showing that ideal speedup in fact can be achieved given that the sequential portion approaches zero for large problem sizes. This is also known as *Gustafson's Law*.

Before moving forward, let us look at the visual representation of the performance metrics we have discussed above taking the overhead into consideration.

Figure 3-19 shows the ideal parallel execution time T_p^{ideal} as a function of p for $p = \{2,4,8,16,32,64\}$. Notice that doubling the number of processors halves the execution time. Also shown, the realistic parallel execution time including overhead (e.g. sequential portion of the program). The constant overhead is also displayed separately. As expected, the execution time converges to the overhead value. For large number of processors and small overhead, the ideal and realistic execution time curves are very difficult to distinguish, as the reciprocal x function is the smallest for large x values, where we are most interested in the results. For this reason, an alternative plot, shown in Figure 3-20, should be used for the execution time where each axes uses a log scale (preferably base 2). This transforms the execution time to a straight line in the ideal case and a curvy one for the realistic cases. The log scale amplifies the small changes at large processor values and makes it visually trivial to spot non-ideal performance (is it a straight line or not?).

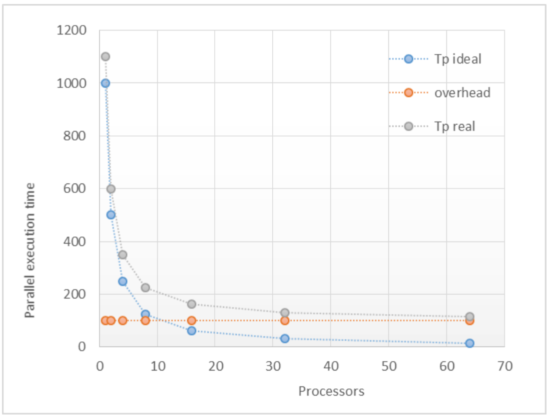


Figure 3-19 Parallel execution time

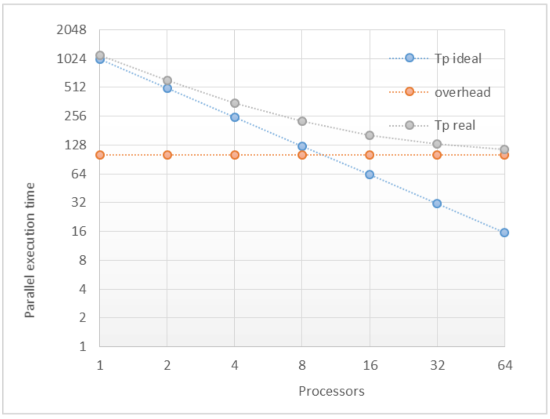


Figure 3-20 Parallel execution time shown in log-log scale

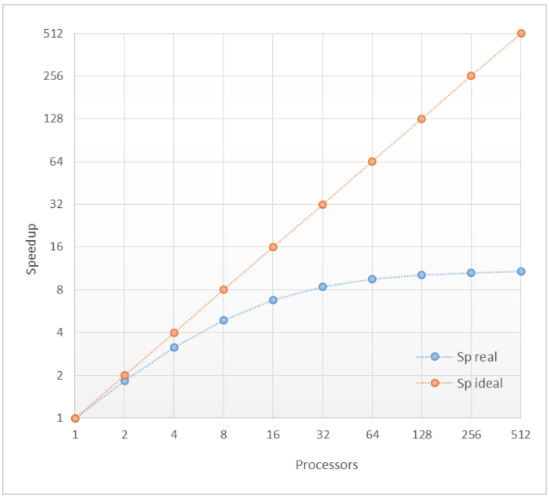


Figure 3-21 Ideal vs. realistic speedup curves.

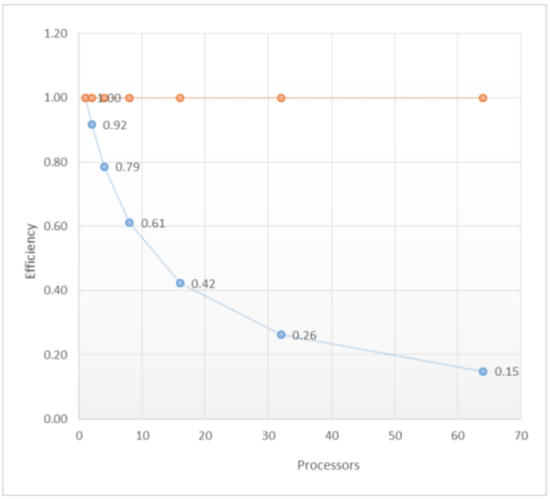


Figure 3-22 Ideal vs. realistic efficiency curves.

Unfortunately, parallel computing in practice is more complicated that we could describe the performance with asymptotic limit analysis. In parallel programs running on multi-processor computers, there are various overheads present that are due to hardware, software and algorithmic factors, for example time lost due to:

- communication and synchronisation among processes,
- waiting for shared resources,
- processors not having enough work to do, or
- not having enough parallelism in the problem.

We can represent the various overhead effects (some being constant, or a function of p or problem size n) as an overhead function $T_O(p,n)$. The use of this overhead function allows us to investigate the performance of our parallel system not only for unrealistically large number of processors but for practical system sizes too. This is crucial as the shape of the overhead function has enormous effects on performance, as we will see below.

Let the parallel execution time include the overhead term as

$$T_p(p,n) = \frac{T_1}{p} + T_O(p,n)$$

The parallel time and the speedup now becomes functions of p and n as well.

Overheads

$$S_p = \frac{T_1}{\frac{T_1}{p}+T_O(p,n)} = \frac{T_1}{\frac{T_1+pT_O(p,n)}{p}} = \frac{T_1 p}{T_1 + pT_O(p,n)}$$

If we assume T_O is constant, taking the limit of the speedup results in Amdahl's Law if T_O represents the sequential portion of the code:

$$\lim_{p\to\infty}\left(S_p\right) = \lim_{p\to\infty}\left(\frac{p}{1+p\frac{T_O}{T_1}}\right) = \frac{T_1}{T_O}$$

Using our earlier example, with $T_O = T_S/10$ – 10% sequential part – the upper bound on the speedup is 10.

If we look at the limit of the speedup using the general overhead function $T_O(p,n)$, the result depends on the order of p in T_O

$$\lim_{p\to\infty}\left(\frac{T_1 p}{T_1 + pT_o(p,n)}\right) = \begin{cases} 0 & if\ a < 0 \\ T_1/T_o & if\ a = 0 \\ p & if\ a > 0 \end{cases}$$

Where a is the order of p in $T_o(p,n)$ as p^a.

The actual speedup function is more informative than the limit, since in practice we run the programs on computers with a limited number of processors. We are much more interested in the shape of the overhead function and the constants that will determine the final time and speedup curves.

Figure 3-23 and Figure 3-24 illustrate the parallel execution time and speedup curves obtained for a parallel program using different problem sizes and an overhead function with complexity $O(p)$; $a = 1$ in $T_o(p,n)$,. We can see that for smallest problem size, the overhead quickly becomes the dominating factor and execution time starts increasing again.

Beware! In this case, adding more than 16 processors to the system will slow down our problem; using more than 128 processors will result in longer execution time than the sequential one!

As the problem size is increased, the effect of the overhead becomes smaller and smaller and the performance tends to the ideal. These effects are even more pronounced on the speedup curves. For the smallest problem size the largest speedup is at *p* = 8, then the system starts to slow down. Increasing the problem size pushes the speedup maximum to larger processor number, *p* = 32, then it also starts to decline. The largest problem size produces near ideal performance up to 32 processors then it starts to saturate but the speedup still increases. The shape of the overhead function did not change; the ratio of the overhead to the useful computation has a dramatic effect on performance. The art of parallel computing is to optimise a system to minimise overheads by carefully matching the parallel architecture, the algorithm, the programming language and the implementation.

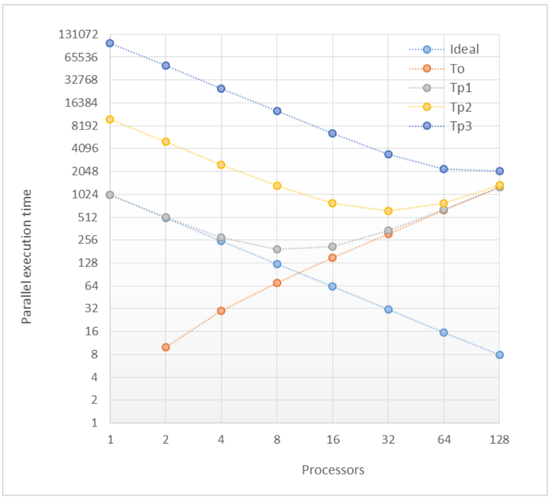


Figure 3-23 Parallel execution time for different problems sizes at linear overhead function.

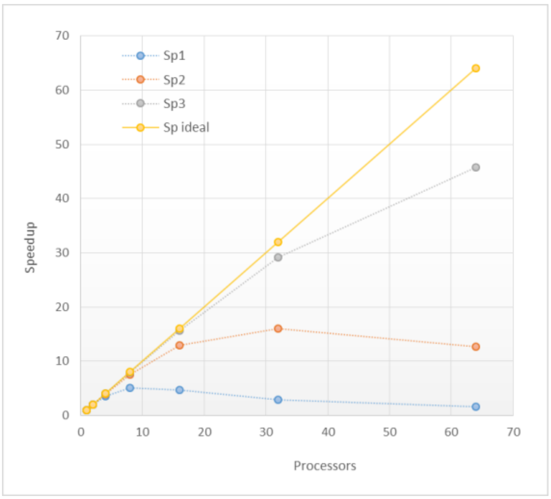


Figure 3-24 Speedup curves for different problems sizes at linear overhead function.

3.6 CREATING AND EXECUTING PARALLEL PROGRAMS

The process of creating a parallel program is very similar to what we normally follow in designing and implementing a traditional sequential program. The main difference is that the programs are more complex and we really need to pay attention to execution speed and performance.

Once a parallelisation strategy is found for the problem we select a parallel programming language that is suitable for describing our solution and can create executable code for our target parallel computer. For desktop computer use, this is normally not a problem, every language can create code for them (although some compilers/runtimes are not well supported on Windows platforms).

The next step is to design whether or not to use external libraries. Much programming effort can be spared if libraries providing reliable and high-performance algorithm implementations can be used in our program and we do not have to develop the entire code from scratch. Linear algebra libraries are good examples for this. However, in some cases, learning a library can be an equally difficult task. Also, there are third-party libraries that provide mediocre performance. For very specialised problems, using libraries is normally not an option.

While most programs can be developed using a simple code editor and command line compiler commands, large production systems are typically developed in Integrated

Development Environment (IDE). If this is a requirement, one must take into IDE support into consideration too. There are a number of development environments in use, probably the most popular ones at the moment are Eclipse and Microsoft Visual Studio. Eclipse supports parallel development via the Parallel Tools Platform plugin.

The most difficult task in a parallel program development is debugging. Our processes might execute on different machines, there might be temporal and non-deterministic errors. These are very difficult to spot and localise in a parallel program. This is the area where modern development environments are indispensable. It must be noted that parallel numerical programs require extra care when developed. The validity and accuracy of the numerical results must be carefully validated to reference sequential programs in order to guarantee that our program computes correct results at high speed – there is not much use in generating results that are useless but very fast!

The final step of the development process is the performance evaluation step. Once the results are validated, the program should be analysed for execution performance. Performance critical parts of the program should be identified and, if possible, optimised to improve performance even further. Performance optimisation is almost an art. It requires deep knowledge of the algorithm, the execution mechanism and the characteristics of the parallel architecture. It is a combination of various software and hardware parameters that will determine the final performance of the program, therefore this task is very frequently a balancing act requiring many experiments. The built-in parallel profilers in state-of-the-art development environments are invaluable tools in this step, indeed.

Programs running in a global address space computer are relatively straightforward to execute. Normally we set the number of processors to be used in the program then run it. Distributed memory systems are more complicated. We need to configure the system, decide how many processors/computers we would like to use in the system and often manually describe how the processes or our program will be mapped to physical processor cores. There are also systems, however, where operating systems take care of the mapping and execution scheduling of processes, so we can draw the conclusion that the variety of options in parallel program execution are much larger than in the sequential programming world.

References

[1] “The History of Computing Project: Cray-1 computer.” [Online]. Available: http://www.thocp.net/hardware/cray_1.htm.

[2] Wikipedia, “Photograph of the Cray-1 supercomputer.” [Online]. Available: http://en.wikipedia.org/wiki/File:Cray-1-deutsches-museum.jpg.

[3] A. R. Larzelere, “Delivering Insight - The History of the Accelerated Strategic Computing Initiative (ASCI),” 2009. [Online]. Available: https://asc.llnl.gov/asc_history/Delivering_Insight_ASCI.pdf.

[4] “Page 2 - Intel’s 50-core champion: In-depth on Xeon Phi | ExtremeTech.” [Online]. Available: http://www.extremetech.com/extreme/133541-intels-64-core-champion-in-depth-on-xeon-phi/2.

[5] “High Performance Computing for Servers | Tesla GPUs | NVIDIA.” [Online]. Available: http://www.nvidia.com/object/tesla-servers.html.

[6] G. Amdahl, “Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities,” in *AFIPS Conference Proceedings (30)*, 1967, pp. 483–485.

[7] J. L. Gustafson, “Reevaluating Amdahl’s Law,” *Commun. ACM*, vol. 31, pp. 532–533, 1988.

4 PARALLEL PROCESSING ARCHITECTURES

Parallel computing is about using multiple processors to reduce program execution time. The processors we intend to use during our computation must form a computing device that is capable of executing parallel programs. In this part of the book, we explore various ways of building multiprocessor computers. We will discuss the architectural, engineering and often economic constraints that influence the type of parallel computer we can create. We will also look at the history of architecture development, which is very important as many ideas that once looked impractical may – and do – reborn later when we encounter a technological paradigm shift. Since the field of parallel computer architecture is vast, we will only cover the most important aspects in this part in order to introduce the reader to the most fundamental issues.

Let us first look at what we need to build a parallel computer. We certainly need processors, memory and some sort of interconnection device that will connect the various parts of the system (typically processors and memory). These are the basic building blocks (components) of any parallel system. Based on the way we interconnect these components we can create two fundamentally different types of parallel computing systems: *shared-memory* and *distributed memory* parallel architectures. Shared-memory systems are often called tightly-coupled systems, whereas the ones using a distributed memory architecture are called loosely-coupled systems. As we will see when we discuss the programming aspects, in tightly-coupled systems, processes communicate via shared variables; in loosely-coupled systems communication is achieved by message passing.

4.1 SHARED-MEMORY COMPUTERS

An archetypical shared-memory multicomputer, as shown in Figure 4-1, consists of a set of processors and one global memory module. The processors are connected to the memory via a common, shared system bus. The system bus in our discussion will always include the address, data and control buses. Each processor accesses the same memory module and each of them reads from and writes to the same address space. This architecture is the obvious extension of the von Neumann type architecture, in which the processor is connected to the memory via a bus. It is not surprising that the first parallel computers are extensions of this very well understood sequential architecture.

4.1.1 The operating mechanism of a single-bus shared-memory multicomputer

When we use multiple processors in a bus-based system, we cannot let processors use the bus at their will. Since the bus can accommodate data transfer for only one processor at any one time, access to the bus should be controlled. Normally, a bus *arbiter* is used for this purpose. Whenever a processor wants to access the memory, it must request the use of the bus from the arbiter by sending it a *request signal*. Based on the request, if the bus is not busy, the arbiter *grants* the right of using the bus to the requesting processor and the memory operation can begin. While the processor-memory data transfer is in place, the bus becomes blocked for other processors; any other processor wishing to use the bus will have to wait. Once the memory operation is complete, the processor releases the bus, and consequently the other arbiter can grant access to the bus for the other processors.

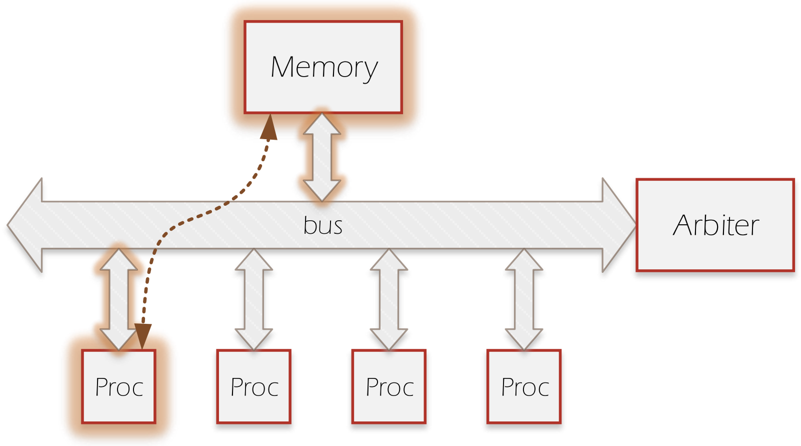


Figure 4-1 The architecture of a typical shared-memory system. Processors access memory via the shared bus.

Bus arbitration methods

There are various common rules based on which the arbiter can decide which processor's bus request to grant when multiple competing requests are received. The most common such methods (arbitration policies) are the following ones [8].

Static daisy chaining — This is a static policy in which priority is attached to system components based on their location on the system bus relative to the bus control unit. The closest component has the highest priority. This scheme is not used frequently in parallel computers; it is more common in industrial systems where additional processors are used to control peripheral devices or provide backup in failure situations.

Fixed time slice (round robin) — In this scheme, fixed-length time slices are allocated to each processor during which it is allowed to use the bus before the next component takes over. The method is executed in a cyclic manner giving a chance for each processor in turns. Each component has equal priority and the service does not depend on the location of the component. This method provides good load balancing but waiting times can be high if the number of processors is too high. This policy is the most commonly used in parallel computers.

Least served first — Under this dynamic scheme, the highest priority is given to the processor that has not used the bus for the longest time. After each bus access operation, a new priority is calculated. This method provides good load balancing with shorter waiting time than the static round robin one. (Interestingly, this kind of priority boost is frequently used by operating systems for giving a chance for low-activity processes in order to avoid starvation.)

First-come, first-served — In this arbitration policy the bus is granted to processors in the order of their requests. The processor placing the request first will receive the bus first. This method is not easy to implement – queues are necessary – but provides the best average waiting times among all schemes. Its drawback is that bad load balance may introduce scheduling problems; high-activity processors will dominate bus usage.

The main attraction of bus-based shared-memory architecture is its simplicity. Since it is the straightforward extension of the type of computer architecture the industry has been using for the past fifty years, the technology is very mature. Creating parallel systems based on the modification of the original sequential architecture hence is relatively simple.

There is, however, one crucial problem with this type of architecture and that is the shared bus. Since processors have to share the bus and only one can use it at a time, all processors except one will always have to wait for the bus. This is what we call *resource contention*. Why is the shared bus a major problem? If we increase the number of processors in the system, processors will have to wait more and more for the bus. For a small number of processors we can, in theory, increase the speed of the bus so it could service more processors within the same amount of time, but the speed of the bus obviously cannot be increased without limits. Consequently, as we add more and more processors to our system, the bus will become more and more busy – in other words the bus becomes *saturated* – and the processors eventually will spend more time waiting than working. The performance of the system will halt, i.e., after a certain threshold, adding more processors will not increase the computing power of the computer.

With this approach, only a very limited number of processors can be used effectively. If we would like to achieve, say, teraflops performance, which requires thousands of today's processors to be used collectively, this is not a viable architecture. In the following sections we explore ways that can improve the performance of bus-based architectures [8].

4.1.2 The cache-based shared-memory multicomputer

One way to improve the performance of bus-based systems is to introduce local cache memories for each processor [9], [10], [11]. In typical program execution cases, if a memory reference is made by a processor, it is very likely that the next reference will be within the neighbourhood of the previous address (e.g., fetching the next instruction for execution, or accessing local variables at the data area). Consequently, by using the cache, the program code and data will be in the cache most of the time and can be accessed locally without contention. This can radically reduce the frequency of using the bus for accessing the shared main memory. The global memory has to be accessed only when there is cache miss (the required reference is not in the cache) or when access to shared variables implementing data exchange among the processes is necessary.

The use of cache memories can dramatically reduce traffic on the shared bus and correspondingly increase the maximum number of processors that can effectively be used in such computers, which in turn will increase the performance of the system. With this modification we can create parallel computers – depending on technological parameters – with processor number in the range of 8-64.

Figure 4-2 illustrates the simplest version of cache memory based parallel systems. Other structures are also possible. One can create a system with local cache plus normal memory per processor and a global memory. It is also possible to go to the extremes and have a cache-only architecture.

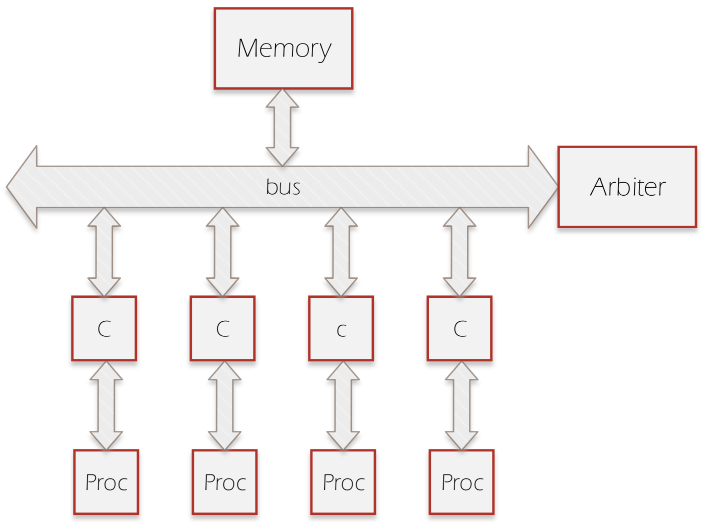


Figure 4-2 A typical shared-memory system with cache memory.

Unfortunately, everything comes with a price. The use of caches introduces another type of problem: the need for cache synchronisation.

4.1.3 Cache synchronisation (the cache coherency problem)

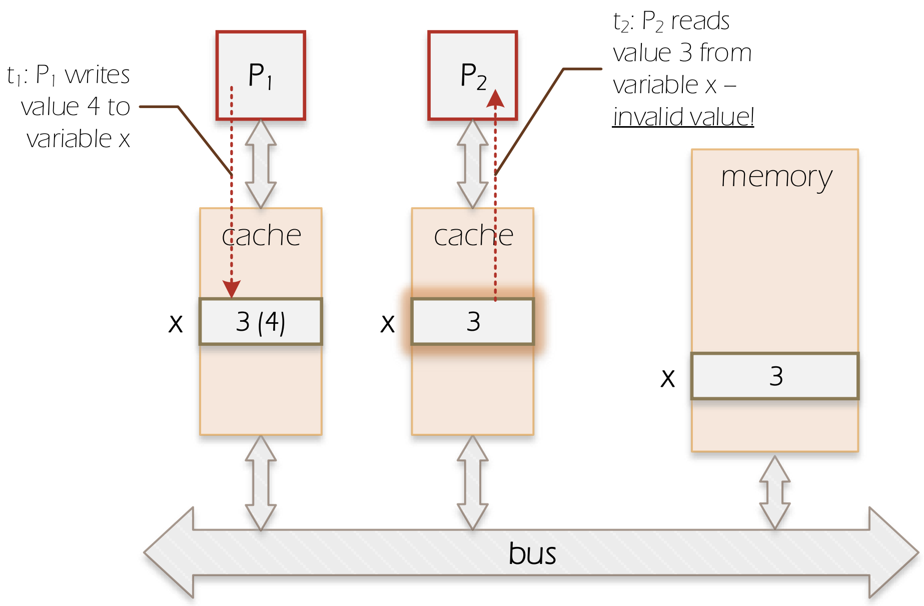
The problem of *cache coherency* and the need for *cache synchronisation*, arises when a shared variable is loaded into several caches and one of the processors modifies that variable in its own cache. Figure 4-3 presents this situation in detail. Assume a variable called x stored in memory with an initial value of 3. Also assume that each processor will cache variable x since they will need it in their section of the program. If at time t_1 P_1 changes variable *x* to 4, the caches in the system will return different values for the same variable (time t_2) as the other caches still store the value 3. In general, if any of the processors of a cache-based parallel computer changes a variable, without proper procedures in place, the other processors would always read the old value, which when used in consecutive computations, could produce disastrous effects!

Cache synchronisation refers to the mechanism that ensures that whenever a variable is changed in one cache, all other caches update themselves by re-reading the main memory. This will result in a consistent, coherent state throughout the system. This, however, sounds much simpler than it actually is in practice [11], [12].

4.1.4 Methods to provide cache coherence

Assume that there is a value to be read from the main system memory at location *x*. A system of caches is coherent *if and only if* a read operation of this value - perhaps a read from a cache if it has been cached - always returns the most recent stored value of location *x*. This is also illustrated in Figure 4-3. Cache coherency can be maintained by cache synchronisation, which includes two crucial steps: (*i*) resolving conflicts in the case of multiple write operations to the same location and (*ii*) maintaining the latest version of data by updating. Conflict resolution is generally provided by hardware solutions. In the followings, we concentrate on the cache update problem and examine the two main forms of cache updating: the

- *write-through* and
- *write-back* methods.



t_1: P_1 writes value 4 to variable x
P_1
P_2
t_2: P_2 reads value 3 from variable x – invalid value!
cache
cache
memory
x
3 (4)
x
3
x
3
bus

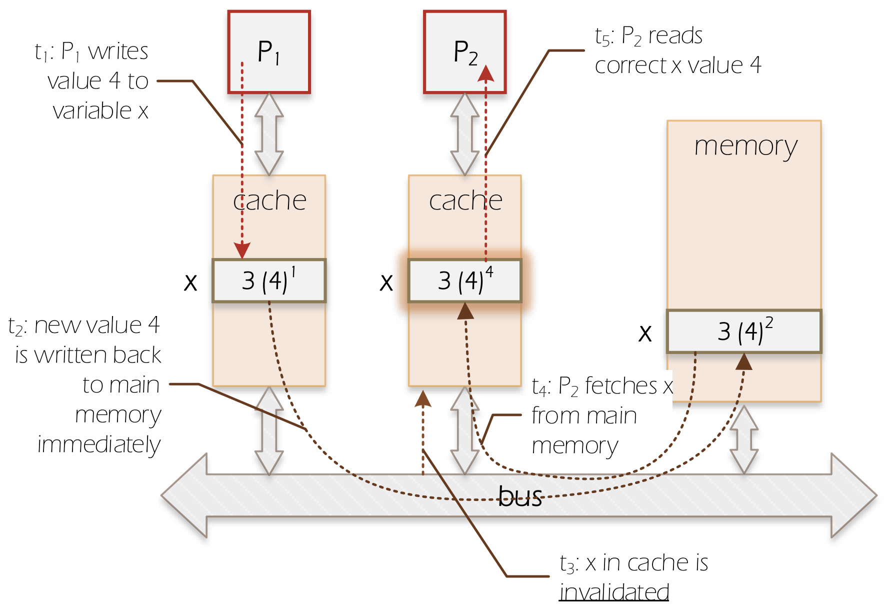
Figure 4-3 Incoherent cache system. P1 changes the value of x in its local cache. Since this change is not reflected in Cache 2 and in the main memory, any other processor reading x will read the value 3.

The write-through approach

One solution for the incoherent cache problem is forcing cache memories to update themselves by invalidation. In the so-called write-through method, every write operation updates the local cache and at the same time it also updates the variable in the main memory and invalidates all other caches in the system.

As shown in Figure 4-4, caches monitor the write address lines and whenever they detect an address that is also held by them, they invalidate the given word in the cache - the processor on the next read operation is forced to read that location from memory. All caches that hold that location invalidate the given data separately.

Although relatively simple to implement, the write-through approach has a limitation which is the high level of bus traffic it generates. Consequently, this method cannot be used for large number of processors.

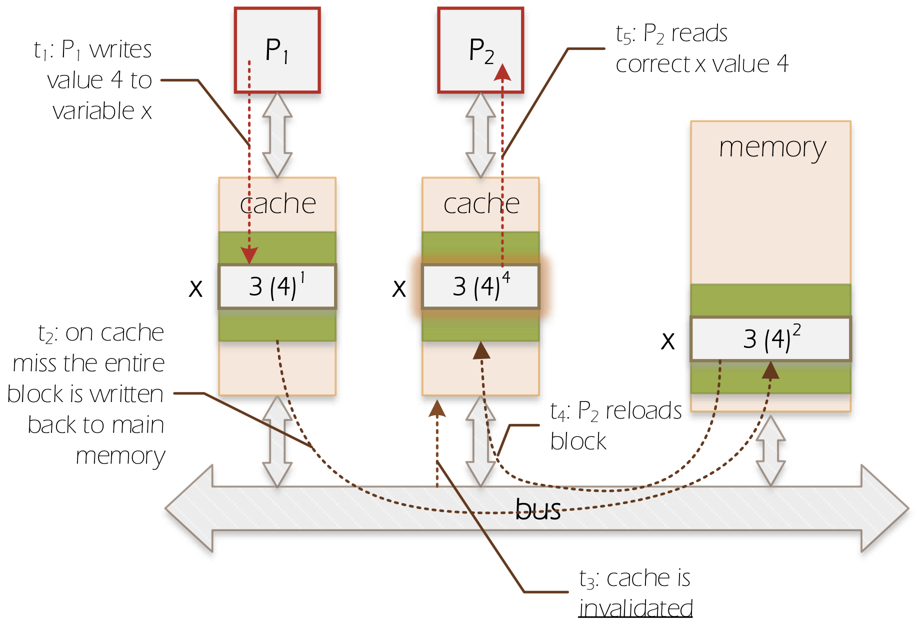


t_1: P_1 writes value 4 to variable x
P_1
P_2
t_5: P_2 reads correct x value 4
memory
cache
cache
x
3 (4)^1
x
3 (4)^4
x
3 (4)^2
t_2: new value 4 is written back to main memory immediately
t_4: P_2 fetches x from main memory
bus
t_3: x in cache is invalidated

Figure 4-4 Illustration of the write-through approach.

The write-back approach

In the write-back method (also known as write-in or copyback), shown in Figure 4-5, the value changed in the cache is written back to main memory only when there is a cache miss therefore the amount of bus traffic is smaller than in the write-through case. Before a new block of data is fetched from the memory on cache miss, the block currently in the cache is written back to main the memory. If another cache changes the value of a location that is also held in other caches, a special hardware notifies all caches and forces them to invalidate the blocks, so the processors are forced to read from the main memory (since the cache miss). The unit of read and write operations (granularity) is always a block of data.



t_1: P_1 writes value 4 to variable x
P_1
P_2
t_5: P_2 reads correct x value 4
memory
cache
cache
x
3 (4)^1
x
3 (4)^4
x
3 (4)^2
t_2: on cache miss the entire block is written back to main memory
t_4: P_2 reloads block
bus
t_3: cache is invalidated

Figure 4-5 Illustration of the write-back approach

4.1.5 Crossbar-based shared memory architectures

As we have seen in the preceding sections, the simple bus architecture provides limited performance due to the single shared bus. Although the introduction of cache memories raises this upper limit, the problem is not yet solved. Is there any other way to create shared-memory systems that do not use the performance-bottleneck shared bus and work efficiently with larger number of processors?

Since the core problem of buses is the single shared data path, a possible improvement is to use such interconnects that provide *multiple data paths*. There are two main forms of achieving this: using *crossbar switches* or *multistage switching networks*.

The crossbar system consists of a set of intelligent switches arranged in a mesh structure, see Figure 4-6. The switches are placed at the intersections of communication lines (running in row and column directions). If a given processor P_i wants to access memory module M_j then the switch S_{ij} is switched on. The beauty of the crossbar is that it provides *p* independent data paths between any processor-memory pair. Problems occur, however, when more than one processors want to access a memory module. This situation is called *contention for memory bank* and to avoid erroneous operation, all except one processor will have to be blocked.

The main drawback of the crossbar interconnection is its cost. Connecting *p* processors and *m* memory modules requires *pm* switches. To reduce contention $m \geq p$, hence at least p^2 switches are required. Since the individual switches are very complex devices (remember, the complete bus has to be switched), the cost and difficulty of manufacturing large crossbars severely limits the maximum size of crossbar-based systems.

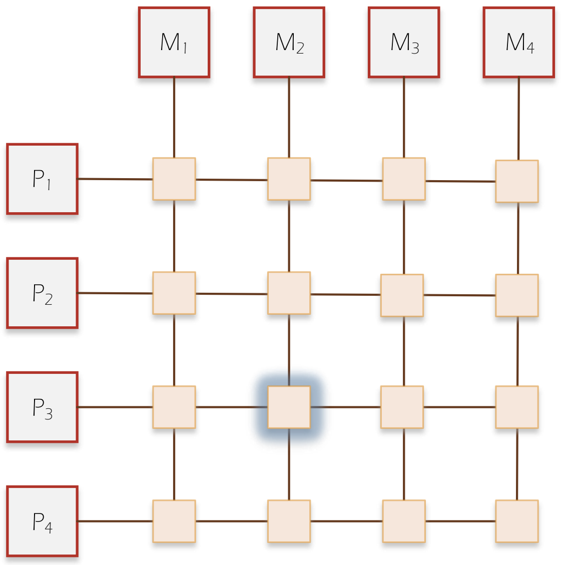


Figure 4-6 The crossbar based shared memory architecture. If the highlighted switch is switched on, processor P_3 can access the memory module M_2, otherwise the path is closed.

4.1.6 Switch-based shared memory architectures

Bus-based systems are relatively simple and inexpensive but cannot be scaled to larger sizes due to the bus saturation problem. Systems using crossbar switches provide higher communication performance and hence could scale up to larger sizes, but the cost of large crossbars is prohibitive. *Multistage interconnection networks* provide a compromise between these two extreme cases. As shown in Figure 4-7, the multistage network consists of a set of cascaded columns of switches.

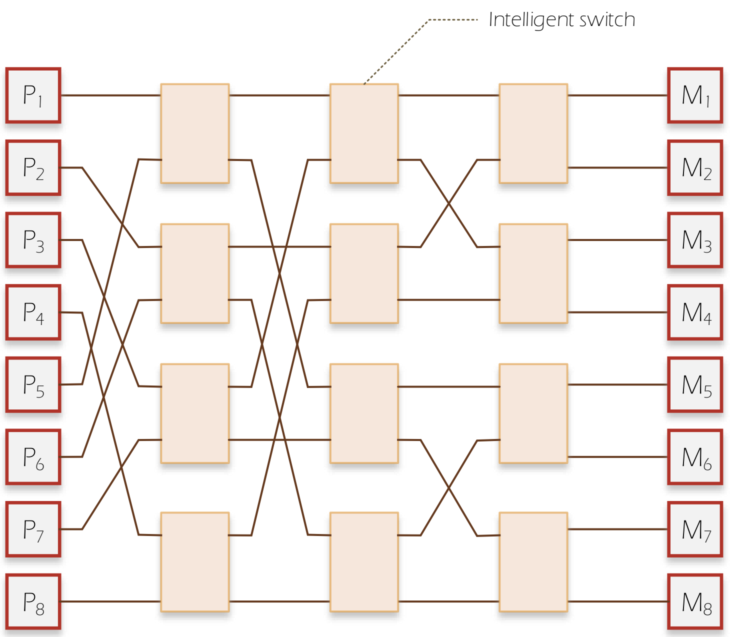


Figure 4-7 Shared memory architecture using multistage switching network. Each stage is a column of switches.

For simplicity, we assume that the number of memory modules *m* is equal to the number of processors *p*. Then, a network of log $p \times p/2$ switches is required to connect the components. This structure provides – in the best case – *p* independent data paths between processors and memory modules. The switching network operates as follows. Each switch has 2 inputs and 2 outputs. Each column of switches (called a stage, hence the name multistage network) connects its inputs to the outputs in some predetermined way. Each switch has four possible configurations: pass-through, cross-over, upper and lower broadcasts, as shown in Figure 4-8.

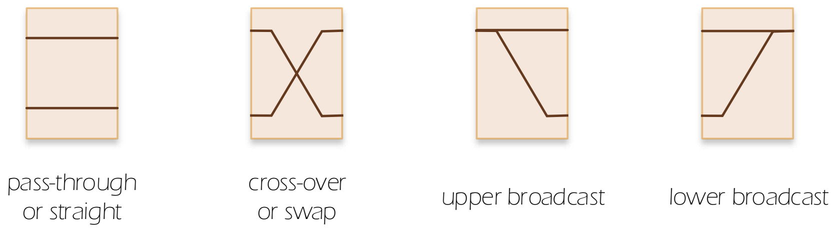


Figure 4-8 Possible switch configurations.

Each switch is intelligent and controlled by a switching function that determines the state of the given switch depending on its position in the network and the path the requested processors is requesting. Figure 4-9 illustrates one possible configuration in which processors P_1 and P_7 can communicate independently on two parallel communication paths. Interested readers can find more details on the control and implementation of the switching elements in [8].

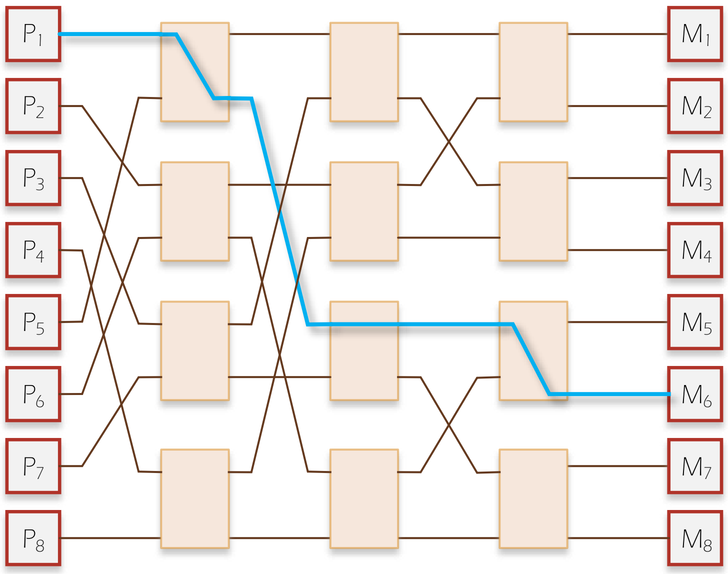


Figure 4-9 The Generalised Cube network topology illustrating communication path from processor P_1 to memory module M_6.

We have seen the ideal situation of multiple processors communicating in parallel. In practice, memory bank conflicts can occur in the system, meaning that two or more processors try to access the same memory module. Similarly, processor-memory data paths may conflict with each other at particular switches. Figure 4-10 illustrates this latter problem. Both of these situations result in blocking, as only one processor can access the conflicting memory module or switch. Switches that frequently block processors due to path conflicts are called *hot-spots*. Several improvements have been introduced to reduce these performance-degrading effects but these are outside to scope of this textbook.

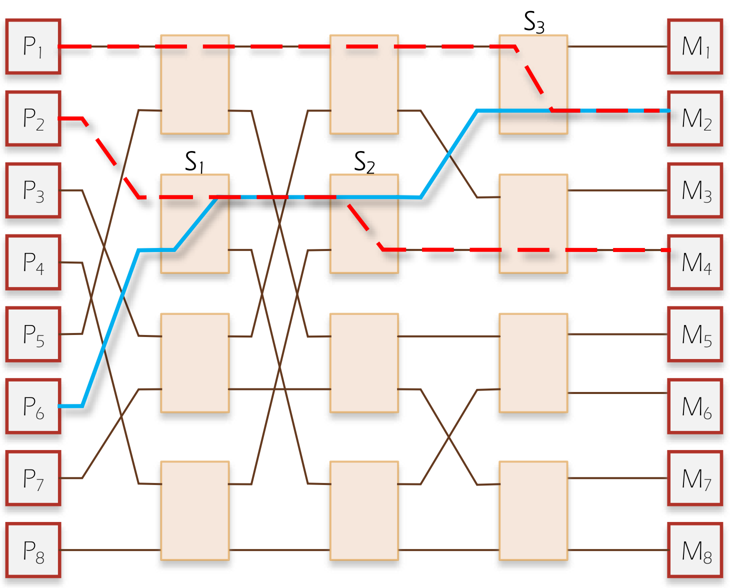


Figure 4-10 Illustration of path conflicts in the switching network. Switches S_1, S_2 and S_3 must block one path requests (in red) until the current communication (in blue) is finished.

Before we continue the discussion with the distributed-memory machines, let us summarise once more what happens when processors in a shared-memory system try to exchange data. Since processors store their data in a common global memory which is visible for all processors, if one writes to a particular address, another processor can read the new value from that address. Exchanging data through memory is called communication using shared variables. There are unwanted consequences of using shared variables in parallel programs; we will discuss these in later in Chapter 0.

4.2 DISTRIBUTED-MEMORY PARALLEL COMPUTERS

The other major alternative approach to arranging and connecting components of a parallel computer is the so-called distributed memory architecture. In this model, processors have only local memory, they are not allowed to access the memory of the other processors. The generic structure is shown in Figure 4-11. Processing nodes are formed of processor-memory pairs and are connected with one another via an interconnection network allowing processors to exchange messages. In the extreme case, when no communication (i.e. data exchange) is necessary among processes, each processor can run the program at its own rate, independently of the others. In this special case, there is no interference in the execution of the processes, consequently, the number of processors in the system can be increased practically without limits.

In practice, this situation rarely happens. Processes running on different processors have to communicate from time to time. What will happen in this case? Since processors need to communicate with each other and without a common global memory, they will send messages to each other. A processor requesting data from another processor sends a message to the target processor. The other processor should then send a message back containing the requested data. This mechanism is called *message passing*. Processors send and receive messages for two distinct reasons: to exchange data and to synchronise their actions/operations.

One can see a clear connection between the type architecture and the method used for implementing data exchange. Due to the lack of common shared memory, shared variables cannot be used. (Although this is not the case with virtual shared memory machines, as we will see later.)

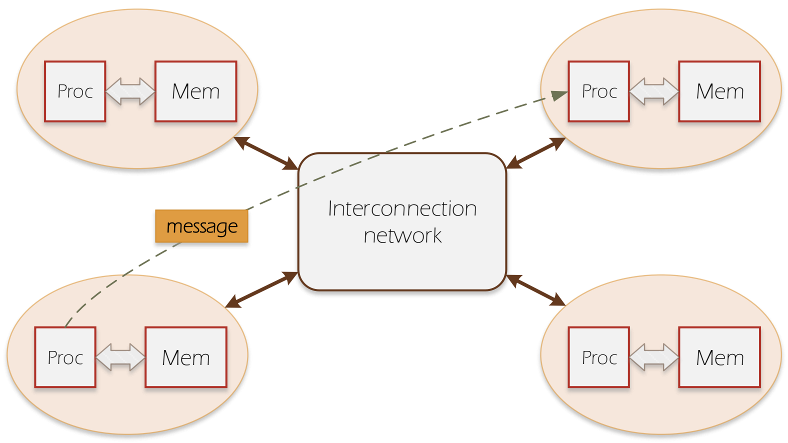


Figure 4-11 A generic distributed memory architecture using message passing. The path of a possible message is shown by the dashed line.

The focus of attention in distributed memory architectures is on the interconnection system, i.e. how processors are connected to each other. We can distinguish two main types of interconnection approaches:

- static point-to-point and
- dynamic interconnection networks.

Static networks represent non-reconfigurable systems with point-to-point communication channels between processors. Dynamic networks are interconnects built from switches or routers that can configure communication paths at runtime. In the following sections we examine the properties of these types of interconnects.

2.2.1 Point-to-point interconnections

In the so-called static, point-to-point interconnection networks we do not use switches but processors are connected to each other by serial communication links. Using these links, we can create different interconnection schemes that we commonly refer to as *topologies*. In the graphs of Figure 4-12, each circle (node of the graph) represents a processor-memory pair while edges represent the communication links.

The first figure illustrates the completely/fully-connected graph/network. A graph is fully connected if there is an edge from every node to every other node. In computing terms, a parallel processor network is fully connected if every processor is connected to every other processor by a communication link. As a result, one communication operation between any pair of processors can be carried out in one time step, as there is a direct connection between them. This would be nice to have, but unfortunately, there are technological constraints that prevent us from realising this topology for large systems. The first one is the number of links required. For fully connecting of p processors, $p(p-1)/2$ links are required. The second one is that – unlike the crossbar where p^2 switches are needed but only one crossbar connection per processor is required – in the fully connected topology the $(p-1)$ links have to be accommodated in one processing node. Even for a small-sized system of, say, 50 processors, this would result 49 communication links per processor. In the past, Inmos Ltd.[5] and Texas Instruments Inc. manufactured processors with 4 and 6 built-in on-chip serial communication links. Today, off-chip communication switches are used to connect processors. Since we cannot practically create fully connected system for more than 6 processors, we must to use partially connected but regular graph topologies. Some of the most common topologies used in parallel computers are shown in Figure 4-12.

The simplest static regular topology is the *line* or *one-dimensional array*. Processors are connected to their left and right neighbours. It is also possible to 'wrap-around' the array to create a *ring* topology that halves the maximum distance to travel in the network.

[5] A former UK (Bristol) based semiconductor company, no longer in business.

Two-dimensional and three-dimensional arrays are extensions of the line topology into further dimensions. These topologies, also called as mesh or grid topologies, are very useful in many numerical algorithms in which dense matrices are used as the main data structures.

The *star* and *tree* topologies belong to the same group of tree topologies, the star being a single-level general tree. The tree topology is most commonly used to broadcast data in a processor network or execute tree-based algorithms.

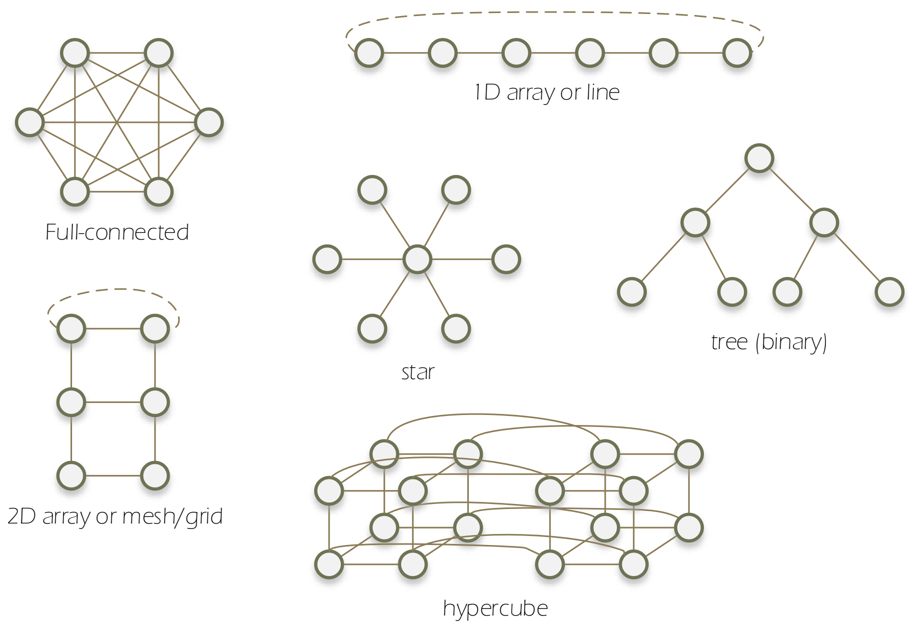


Figure 4-12 Typical interconnection topologies.

The hypercube topology is an *n*-dimensional cube in which the number of processors is $p = 2^d$, where *d* is the dimension and each processor is connected to exactly *d* neighbours. The hypercube topology was very popular in early parallel computers (see Connection Machine 2) because of their nice mathematical properties. Its recursive nature and the ability to embed in them all other topologies elegantly were useful features. They are not so good however when it comes to manufacturing.

2.2.2 Comparing static topologies

There is no topology that is better in every way than the others. Some is good for one particular type of problem while some others are good for other types of problems. They have to be selected based on the particular application. Fortunately, we can use a set of common metrics to compare known topologies and based on them select the one that is optimal for the algorithm in question. The metrics, not surprisingly, come from graph theory:

Degree — the number of communication links per processor. This property is important from scalability and manufacturability reasons. Processors in practice can have only a very limited number of links. If the degree is not constant as we increase the size of the system, all processors must be replaced with ones having more links.

Diameter — the maximum distance a message has to travel between any two processors. This property affects the communication performance of the system. The further a message has to travel, the more time it will take.

Arc connectivity — the minimum number of links that must be removed from the network to break it into two disconnected parts. This property is an indication of the parallel communication paths in the network. The higher this number, the more links (data paths) can be used at the same time without contention.

Bisection width — the minimum number of links that must be removed to partition the network into two equal halves. This metric gives us information about the number of communication channels between the two halves of the system. This is important in certain classes of algorithms that require large amount of global communication operations (e.g. in FFT, sorting, matrix-matrix multiplication).

Cost — the number of links in the network. This property provides a measure of the complexity of the network in terms of links. The more links we need, the more expensive and complicated our system will be.

4.2.1 Routing and communication in static networks

Static interconnection systems are constructed without switches; processors are connected directly by communication links. As a consequence, a message sent to a non-adjacent processor has to travel through intermediate processors (nodes). Communication in message-passing systems can be a major source of overhead. The time that it takes for one message to arrive to its destination can be substantial. This time is called the *latency* and it depends on system parameters as well as the routing mechanism in the system. There are three main factors that determine the time of sending a message over a communication link. The time it takes from the start of the send instruction till the first byte is sent on the physical link is called the *startup time*, T_s. This time is always spent in communication regardless the length of the message and depends on the number of instructions required to implement the send/receive operations as well as the clock frequency of the processor. The time to send one byte over the physical link is called the *per-byte transfer time*, T_b. this time depends on the speed (bandwidth) of the communication link. In certain routing systems, there is a third parameter which represents the time spent with transferring the message in an intermediate node; this is called the *per-hop time*, T_h.

Table 4-1 Topology comparison table for p-processor systems

Topology	**Degree**	**Diameter**	**Bisection width**	**Arc connectivity**	**Cost (no. of links)**
Completely-connected	$p-1$	1	$p^2/4$	$p-1$	$p(p-1)/2$
Star	$p-1$	2	1	1	$p-1$
Binary tree	3	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	2	$p-1$	1	1	$p-1$
Ring	2	$\lfloor p/2 \rfloor$	2	2	p
2-D array	4	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D torus	4	$2\lfloor \sqrt{p}/2 \rfloor$	$2\sqrt{p}$	4	$2p$
3-D array	6				
Hypercube	$\log p$	$\log p$	$p/2$	$\log p$	$(p \log p)/2$
Wraparound *k*-ary *d*-cube	$\log p$	$d\lfloor k/2 \rfloor$	$2k^{d-1}$	$2d$	dp

Store-and-Forward routing

The simplest form of routing is the store-and-forward mechanism. Assume the processor P_i needs to send a message to P_{i+3} in a line topology as shown in Figure 4-13 by the dashed arrow. Since there is no direct connection between these two processors, the message will be sent via intermediate processors, P_{i+1} and P_{i+2}. The message then will arrive in three steps. First we send it to P_{i+1}, where it is stored temporarily. P_{i+1} then forwards the message to P_{i+2} that also stores the message then sends it to P_{i+3}. Note that the entire message must be received and stored in an intermediate buffer before it is sent to the next node, just like in an email delivery system.

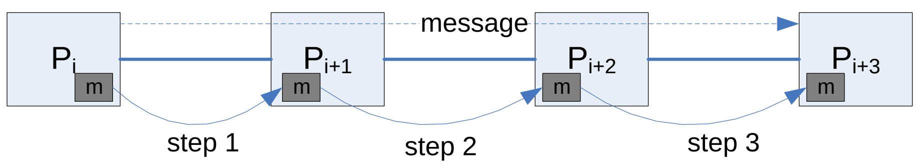


Figure 4-13 Store-and-Forward routing

The time to send the message over one link in this routing scheme is

$$T_{link} = T_s + kT_b$$

where k is the number of bytes in the message. Since we need several steps to reach the destination processor, the overall communication time in this example is three times more; in a general case, the link transfer time multiplied by the diameter d of the network (in this case $d = p - 1 = 3$),

$$T_{comm} = d(T_s + kT_b)$$

This is not an efficient routing method as the communication time is proportional to the diameter of the system which results in large communication overheads for some topologies. Moreover, the startup time is typically much larger than the per-byte time, therefore for small messages, T_s can easily dominate the communication time. In clusters, for instance, communication startup time is typically in the range of 1-100 μsec, depending on various hardware-software factors, whereas per-byte time is in the range of nanoseconds (assuming gigabit links).

Cut-Through Routing

A more efficient way of sending the message is to receive a channel from source to destination and to send it without storing. This is called circuit switching. The problem with this approach is that this reserves the channel and many intermediate routers. A better alternative is virtual cut-through routing (or simply cut-through routing) in which a message is divided into packets. These packets are forwarded to the router without buffering if a free communication link is available. If there is no link to be used, the packet is stored. The advantage of this method is that only packets must be stored in the router, not the entire message. The message has a header that contains the destination processor ID. A node receiving the message will first receive the header, examine it, and if the message is for others, it will redirect the incoming data stream to the outgoing link. The total communication time with cut-through routing is calculated as the sum of the message startup time, the header transfer time and the message transfer time:

$$T_{comm} = T_s + dT_h + kT_b$$

where T_h is the header per-hop time for one packet, d is the diameter and k is the message length.

A variation of cut-through-routing is *wormhole routing*. In this scheme, the message is split into many small parts (called *flits*), and as an intermediate node receives one part, it forwards it simultaneously with receiving the next part. Instead of storing packets, routers only need to store one flit before retransmission. This results in a very efficient form of message transmission in which the communication time will only depend on the length of the message and will be independent of the length of the path. Figure 4-14 illustrates this concept with a message that is divided into four parts, 1,2,3 and 4. Part 1 is sent to P_{i+1} and when part 2 is sent P_{i+1}, P_{i+1} will send simultaneously the already received part 1 to P_{i+2}. Each intermediate node works the same way, and the movement of the message resembles the way a worm is moving in the soil.

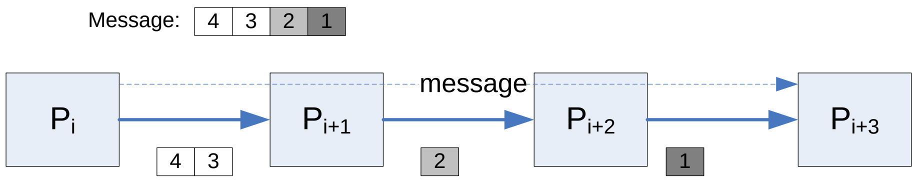


Figure 4-14 Cut-through routing

The time to send the message over one link with wormhole routing scheme is

$$T_{comm} = T_s + dT_h + kT_b$$

where T_h is the flit per-hop time and k is the message length. Figure 4-15 compares the two types of routing schemes visually.

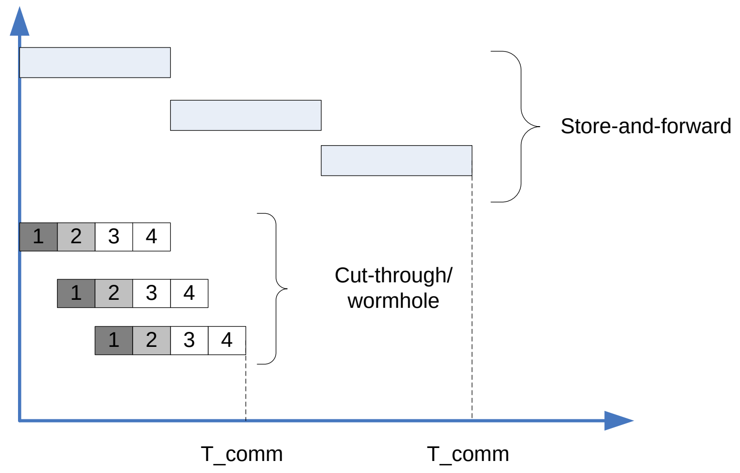


Figure 4-15 Comparing Store-and-Forward and cut-through routing performance

2.2.3 Dynamic Interconnection Systems

The advantage of static topologies is that they are relatively simple to construct but their main disadvantage is that each topology suits only a class of problems. If we need a general purpose computer, we need to support all classes of parallel problems which potentially require very different patterns of communication. To server this need, dynamic interconnects have been introduced in distributed memory architectures as well.

The first possible interconnect is the *crossbar* we already discussed in Section 4.1.5. If we change the scheme from connecting memory modules with processors to connecting processing elements or nodes to processing elements, we have a distributed-memory architecture with a crossbar interconnect that provides higher performance than the shared bus. The advantage of this crossbar over the shared-memory version is that in this case we only need to switch serial communication links instead of entire buses, hence the manufacturing complexity and cost of the crossbar is much lower.

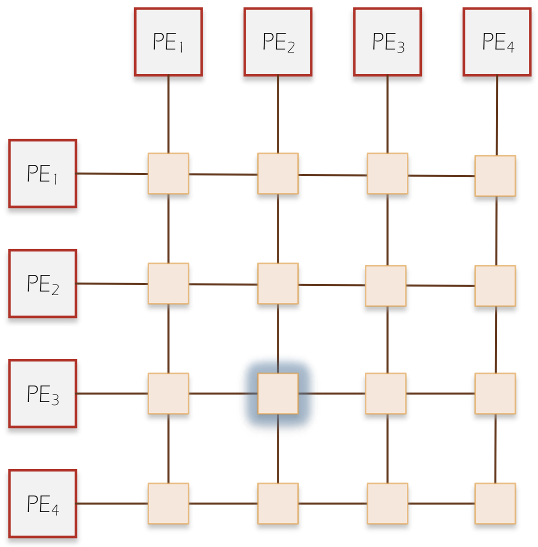


Figure 4-16 Crossbar interconnection in message-passing systems

Continuing with the modification, we can also replace memory modules with processors in the multi-stage switching network also discussed in Section 4.1.6. This system has the same advantages as in the shared-memory case – $O(p \log p)$ switches – but again, by switching only serial links, the cost of this interconnect is much lower.

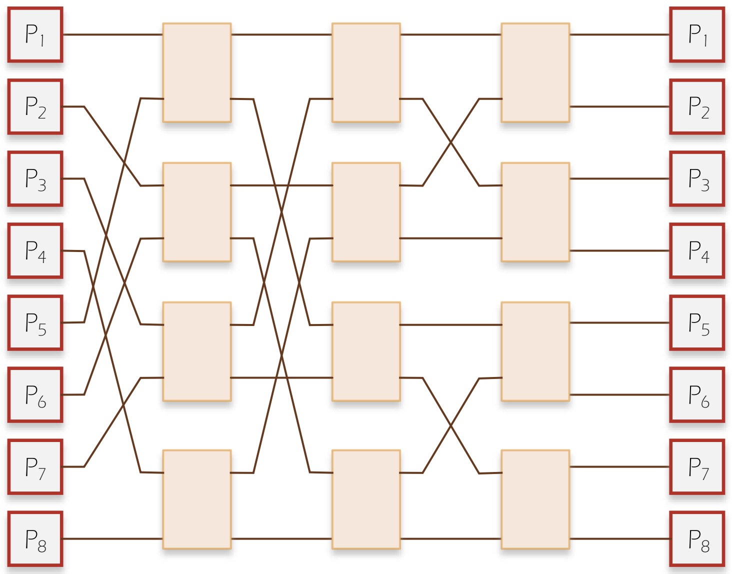


Figure 4-17 Multi-stage switching interconnection network for distributed memory systems.

Routing networks

Current parallel and supercomputer technology uses a combination of static and dynamic interconnects. The typical approach is to connect a router/switch network according to a fixed topology and use dynamic, adaptive wormhole routing to send messages to their destination. This is a necessary technological compromise as communication links are not integrated into current mainstream processors. Processors, *P*, are connected to network interfaces or routers, *R*, as shown in Figure 4-18 that perform the required communication without processor involvement. In our example, a 2D torus router interconnect is shown. The 3D version of this topology is common in current supercomputers, used for instance in the IBM BlueGene[6] computer.

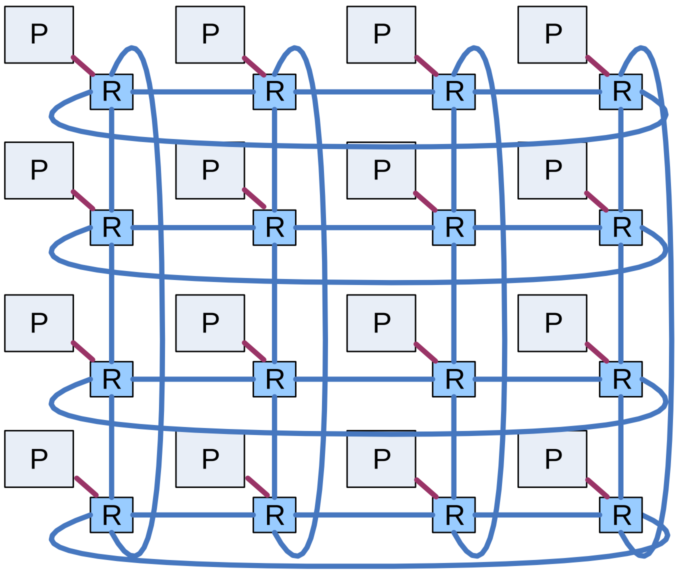


Figure 4-18 Typical 2D routing fabric found in modern multiprocessor computers

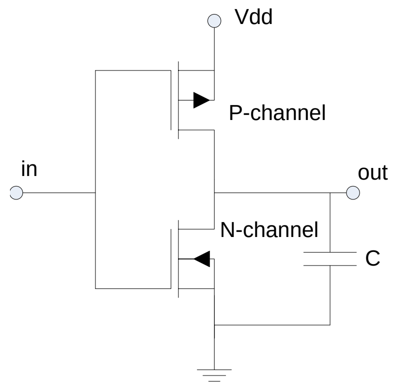
4.3 Multi-core processors

Till about the year 2005, the way to increase microprocessor performance was to increase clock speed [13]. The years of optimism over ever faster processors[7] was over when Intel failed to launch the promised new 4GHz Intel Pentium processor [14]. The project clearly showed the technological limits of increasing frequency indefinitely. With 150W to dissipate, this chip was just too hot to operate reliably. Microprocessor technology has hit what we call the *power wall*.

[6] For further details see http://www.alcf.anl.gov/sites/www.alcf.anl.gov/files/IBM_BGQ_Architecture_0.pdf

[7] For a rather amusing prediction from Intel back in 2000, consult: http://www.geek.com/chips/intel-predicts-10ghz-chips-by-2011-564808/

To understand the power wall effect, we need to step back a bit and look at semiconductor fundamentals. Current microprocessors are built using CMOS technology. The key element of the processors is the CMOS inverter, shown in Figure 4-19, that basically works as a binary switch. This inverter has the nice property that it does not consume power when not operating. It only needs power when switching from one state to the other. The power requirement in this transient operation is called the dynamic power of the inverter. Since processors are constructed by millions of these inverters, the overall chip power consumption is dominated by the dynamic power consumption. Unfortunately, as the size of the components is getting smaller and smaller, CMOS components do not behave as ideally as they used to! Due to a phenomena called sub-threshold leakage[8] and gate tunnel effects, there is an increasing static power consumption as well.



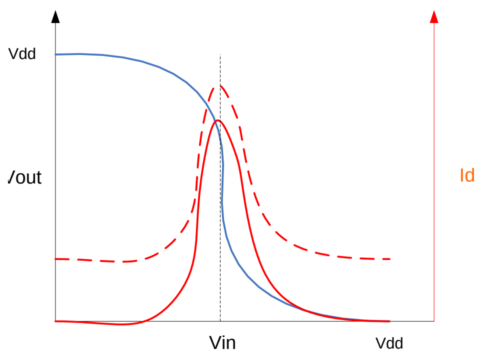


Figure 4-19 The CMOS inverter

Figure 4-20 Switching performance in ideal (solid red line) and real (dashed red line) CMOS inverter. The difference is due to leakage current.

[8] http://en.wikipedia.org/wiki/Subthreshold_conduction

The overall power consumption of the chip P is expressed as

$$P = CfV^2 + P_{static}$$

where C is the capacitance of the transistor, f is the clock frequency and V is the operating voltage of the chip. Increasing clock frequency obviously increases power consumption and the dissipation needs of the chip. The increasing operating temperature has a side effect of increasing static leakage current, hence static power. Static power is also increased by manufacturing size reduction. While about 15 years ago static power was completely negligible, up to around 40% of the power consumption of current processors is attributed to static power. This has put a limit on further increase of single-core processor performance.

To keep power consumption and operating temperature down, we can decrease the chip voltage and the frequency. CPU input voltage is already low, currently around 1 Volt, which is very near to the threshold voltage of transistors; therefore we have limited possibilities here. The only other option is to reduce frequency which results in immediate performance decrease!

It has been shown [15] that by lowering the clock frequency by 20%, approximately 50% decrease in power consumption can be achieved while the performance only dropped to 87%. Consequently, adding two such cores into one chip, we can achieve 174% of the original performance using the same amount of power as before. But this performance increase can only be utilised by parallel programs! Running a sequential program on one of the cores at lowered frequency will decrease performance. From 2006 to 2010, the entire microprocessor industry had switched to multi-core technology. Today, even our mobile phones come with 2-core processors. Quad-core chips are becoming common in desktop computers while 6- and 8-core processors are appearing in servers. This brings us to the most important architectural statement of this section.

Parallel computing is now mainstream.

Single-core computing is over, anywhere we look – from mobile to supercomputer systems – we find multiple, often heterogeneous processing architectures. To harness this power, however, we need parallel software. Unfortunately, this is easier said than done, as we will see in the subsequent parts of this book. Yet, the next two decades will be an extremely exciting and interesting period in the history of computing as we move from the past sequential world to the new parallel one.

4.3.1 The evolution of multi-core processors

Multi-core technology is a relatively new field within processor and computer architecture design. Engineers and scientists are constantly looking for new and improved methods to increase the performance of the processors and at the same time reduce their power consumption and manufacturing cost. Thanks to the experimentation, a number of clear architectural trends are now emerging. Figure 4-21 shows the simplest extension of the single-core chip design. The multi-core is created by placing multiple copies of a single core processor on a single chip. The cores connect to the system bus and communicate via the shared memory.

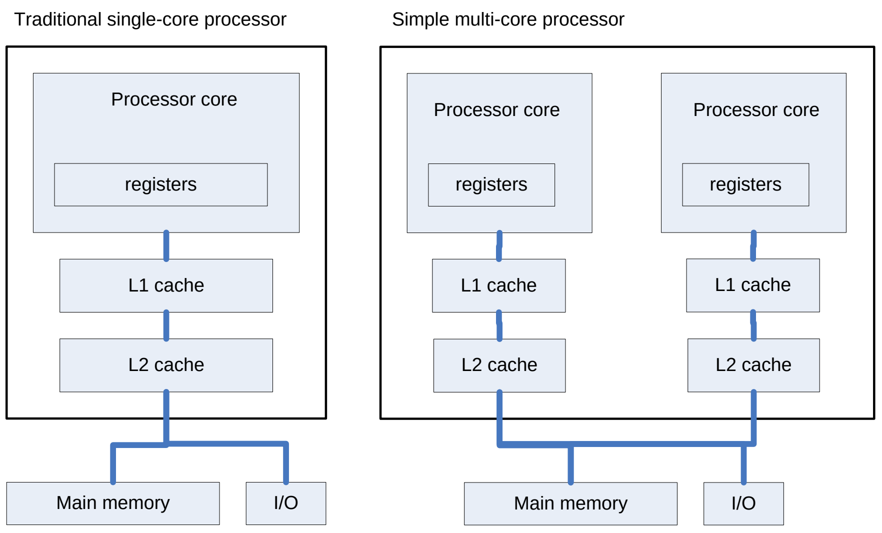


Figure 4-21 Single and multi-core architecture.

A further variant of this design is the shared-cache multi-core architecture show in Figure 4-22. The Level 2 cache is shared among the cores to improve performance (in some processors there is a level 3 shared cache on chip as well). Multi-core processors with cache memories suffer from the same cache coherency problem we discussed earlier; consequently, processors include on-chip hardware support to provide cache coherency. The shared L2 cache simplifies cache coherency hardware while improving performance due to better cache utilisation and higher memory bandwidth.

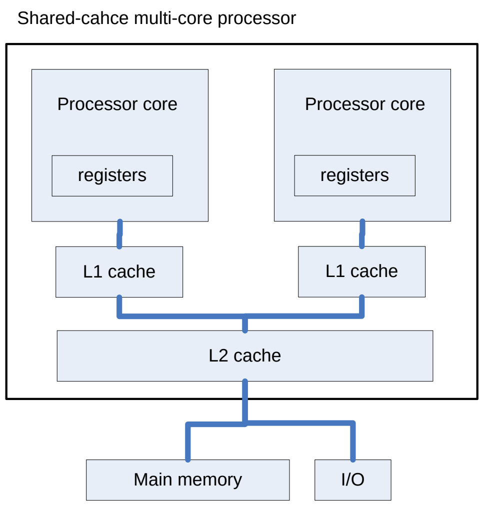


Figure 4-22 Shared cache multi-core architecture.

Having looked at why the processor industry moved towards multi-core processors and what the main architectural issues are in multi-processor systems, let us continue with a quick look at the current situation in the multi-core processor world. We will overview developments and representative multi-core chips of key market areas.

4.3.2 Multicore technology at Intel

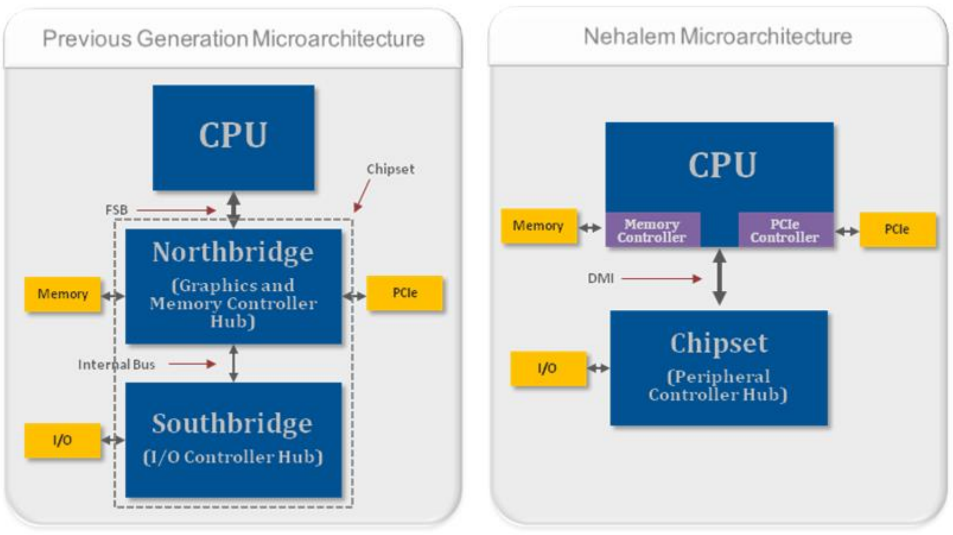
Intel created a roadmap for introducing multi-core processors before 2005. In the past ten years, they have successfully executed the plans set out in that roadmap. Multi-core Intel processors are mainstream in server/desktop/notebook computers.

The first multi-core chips at Intel were based on the architecture shown in Figure 4-22. Representative processor families using this architecture is the Core 2 Duo, Quad and early Xeon processors. The successor Nehalem architecture introduced interesting architectural changes. From the traditional system bus architecture (Front Side Bus) the memory and PCIe controller was moved in the processor core, see Figure 4-23. For a single core this meant faster access to memory and other system components. In addition,

Intel introduced an on-chip point-to-point interconnect system QPI (QuickPath Interconnect[9]) for the cores that for multi-core processors, enables direct core communication, as well as creates cache coherent distributed shared memory architecture on chip. The details of this architecture can be seen in Figure 4-24. This interconnect provides much improved data

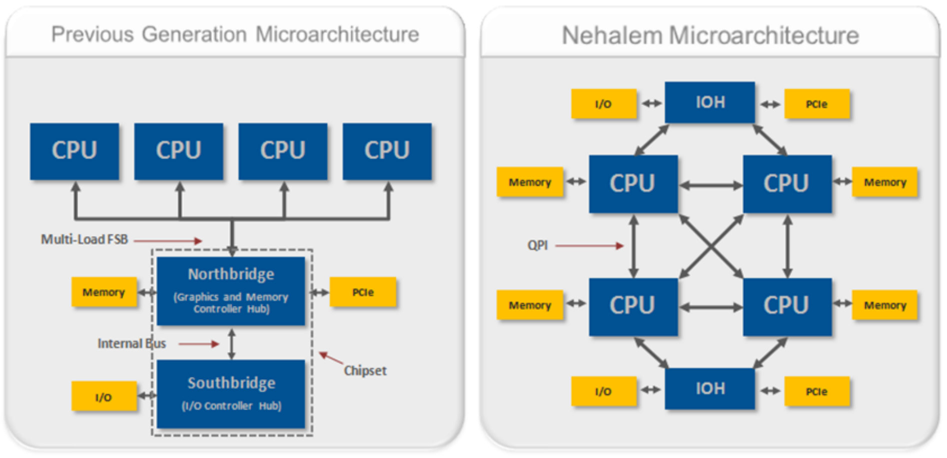
[9] For further details, see: http://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect

communication performance. Variants of the Intel QuickPath Architecture [16] appear in the successor architecture generation Sandy Bridge, Ivy Bridge and Haswell too, providing the basis of high-performance Core i5, i7 and Xeon processors.



Previous Generation Microarchitecture
CPU
Chipset
FSB
Northbridge (Graphics and Memory Controller Hub)
Memory
PCIe
Internal Bus
Southbridge (I/O Controller Hub)
I/O
Nehalem Microarchitecture
CPU
Memory
Memory Controller
PCIe Controller
PCIe
DMI
Chipset (Peripheral Controller Hub)
I/O

Figure 4-23 The new memory controller and PCIe controllers introduced for processor cores in the Nehalem architecture[10].



Previous Generation Microarchitecture
CPU
CPU
CPU
CPU
Multi-Load FSB
Northbridge (Graphics and Memory Controller Hub)
Memory
PCIe
Internal Bus
Chipset
Southbridge (I/O Controller Hub)
I/O
Nehalem Microarchitecture
I/O
IOH
PCIe
Memory
CPU
CPU
Memory
QPI
Memory
CPU
CPU
Memory
I/O
IOH
PCIe

Figure 4-24 The move from the shared bus to the new core-oriented interconnection system introduced in the Nehalem architecture[10].

[10] Image source: Texas Instruments White Paper, Top Eight Features of the Intel Core i7 Processors for Test, Measurement, and Control, http://www.ni.com/white-paper/11266/en/

4.3.3 Multicore technology at Sun/Oracle

Sun Microsystems (now Oracle) was among the first ones to embrace multi-core technology. Their successful Niagara chip demonstrates how servers could benefit from multi-core processors. The Niagara 1 chip (2005) had 8 cores with hardware support for executing 4 threads per core. The successor Niagara 2 processor had 8 cores with executing 8 threads simultaneously per core i.e., a total of 64 threads per chip (2007). The Victoria Falls and Rock processors (with 16 cores) followed as improvements before Oracle bought Sun Microsystems. After the Sun's acquisition Oracle took over the developments and continued the improvement of the multi-core, multi-threaded SPARC processors. Their most recent processors are SPARC T5 and M5 models [17]. The T5 processor is a 16 core processor with each core capable of executing 8 threads in parallel. Each processor, hence, can execute 128 threads at the same time.

Each processor includes an internal 8x9 crossbar switch to create a cache-coherent shared L3 cache memory. It also includes another 4x4 crossbar switch that connects the memory Coherence Units to the 7 external direct, point-to-point interconnect links that can be used to interconnect up to 8 processors in a system, creating a total of 128 cores capable of executing 1024 threads at the same time. Instead of high-performance numerical applications, the SPARC processors are optimised for typical server application workloads, such as database query, transaction processing or single-threaded batch operations.

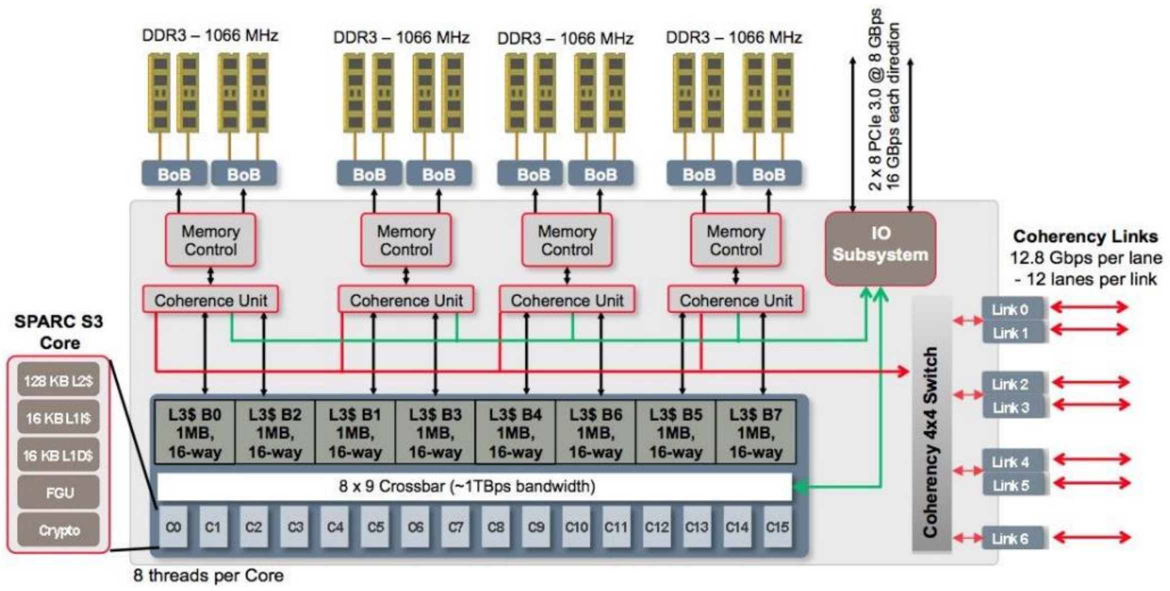


Figure 4-25 Internal architecture of the SPARC T5 processor.

4.3.4 Tilera Corporation

Another approach followed by Tilera Corp. is to aim at larger cores from the start and create an architecture that is more scalable, even up to tens or hundreds of cores. To achieve this, they decided to use an on-chip mesh interconnect instead of a bus. A 2D mesh of switches (iMesh[11]) connect cores and cache modules attached directly to each switch node in a cache-coherent manner. In the early days they used special purpose cores for signal processing applications but in the most recent processor families the company now uses general purpose 64-bit cores. The block diagram of the TilePro64 and the TILE-Gx8072 processors is shown in Figure 4-26 and Figure 4-27, respectively.

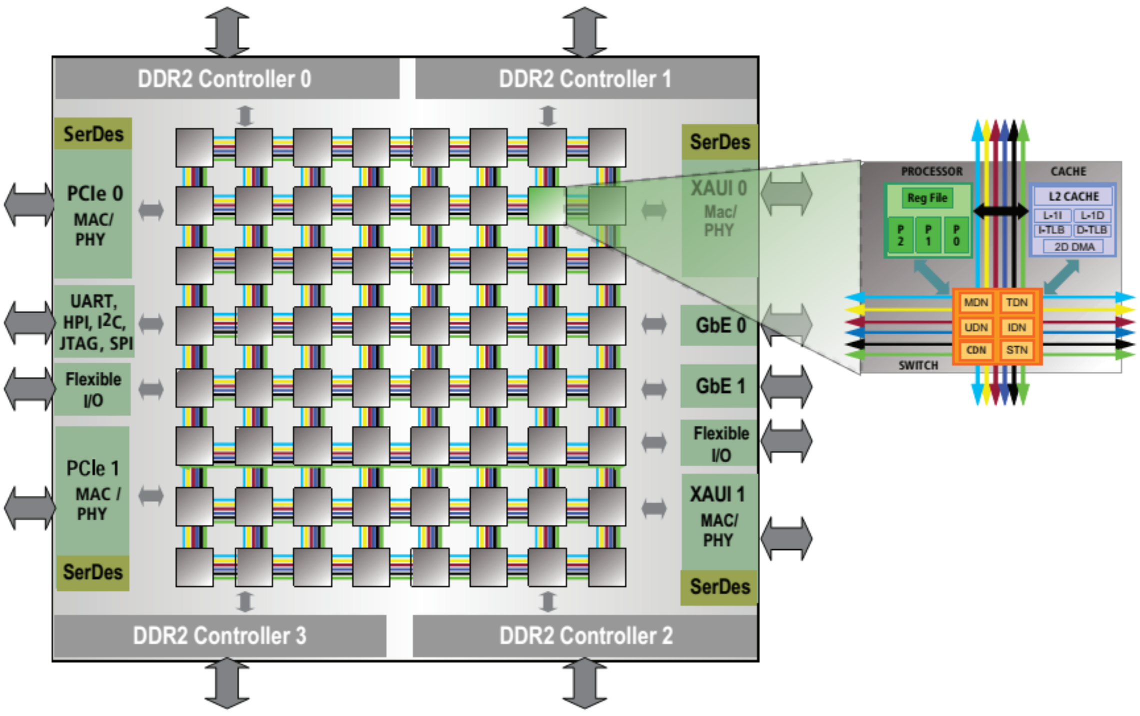


Figure 4-26 Block diagram of the Tilera TilePro64 processor

[11] http://www.tilera.com/products/processors/TILE-Gx_Family

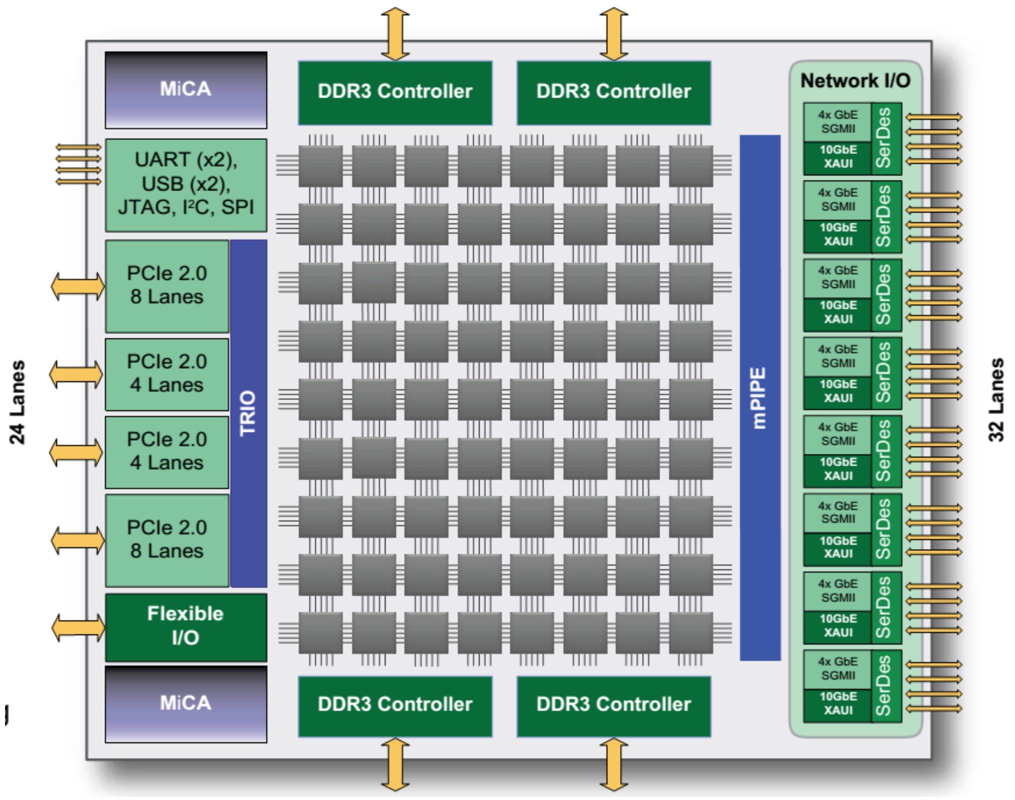


Figure 4-27 Block diagram of the Tilera TILE-Gx8072 processor

4.3.5 NVIDIA CUDA architecture

A parallel computing system that we cannot leave out from the architecture discussion is the NVIDIA GPU (Graphics Processing Unit) computing system. The NVIDIA Corporation has chosen a very different path for developing multi-core processors. They started from the hardware-accelerated pipelines used for 3D games and graphics applications. These pipelines, as they evolved, has become programmable processors and since 2005 became one of the biggest success story in high-performance computing. The key to their success is that they created an architecture using many simple cores that are equally suited to graphics tasks and data-parallel applications.

The details of the graphics hardware and GPU evolution can be found on the web, here we only discuss the current Kepler processor architecture. Each graphics processor can contain a number of Streaming Multiprocessors (SMX), each SMX having 192 cores. The top-end K40 GPU has 15 SMX units, providing a total of 2880 cores and 4.29 teraflops single-precision and 1.43 teraflops double-precision peak floating point performance.

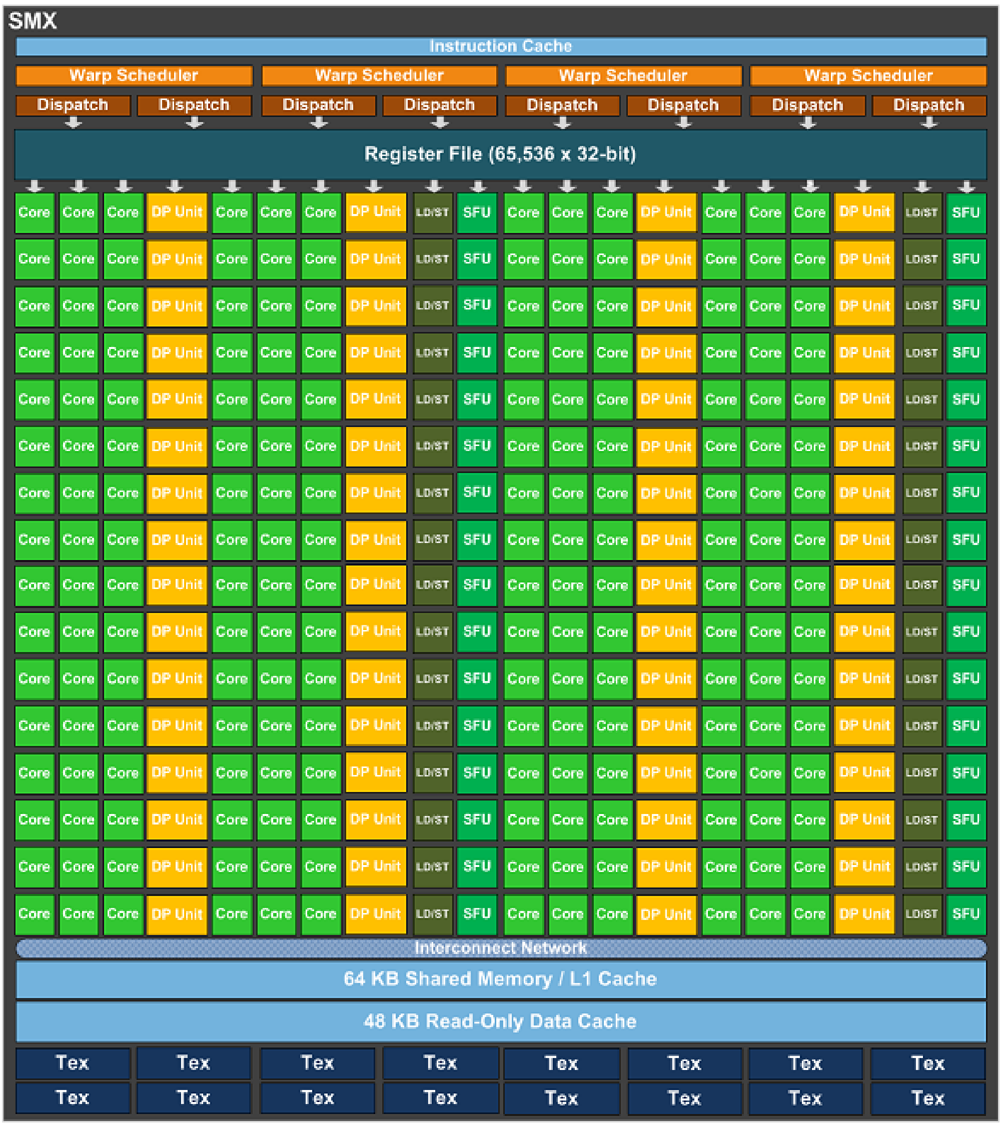

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Figure 4-28 Internal architecture of the NVIDIA Kepler Streaming Multiprocessor.

To summarise the various multi-core approaches, we can see that multi-core technology rules current computing hardware developments. It is interesting to note, that specific types of multiprocessor architectures and techniques used in the past decades are finding their way into current and future multi-core chips. Since semiconductor manufacturing is expected to pack even more components to chips (10 nm or less manufacturing technology is possible to achieve within years), the number of cores may increase to the order of thousands and we will see a variety of homogeneous and heterogeneous multi-core processors in the near future.

4.4 Taxonomy of Parallel Computers

In this chapter, we have seen different approaches to constructing parallel computers. We can create shared-memory architecture, or a point-to-point system with perhaps virtual shared memory on top of it, etc. Is there a way to make order in this architecture jungle? If we need to compare things, there must exist taxonomies, i.e. systematic ways to classify them. In the final part of this chapter we illustrate how parallel computers can be classified depending on key properties of the systems.

Classification by control (Flynn's taxonomy)

One of earliest classification was Flynn's which differentiates the computers based on their control properties [18]. There are four classes of systems in his taxonomy:

- SISD (Single Instruction stream, Single Data stream)
- SIMD (Single Instruction stream, Multiple Data stream)
- MISD (Multiple Instruction stream, Single Data stream)
- MIMD (Multiple Instruction stream, Multiple Data stream)

The SISD machine is the traditional sequential machine. One instruction sequence is executed on one stream of data. In the SIMD model we execute the same single instruction sequence on multiple data sets. This model is very important in parallel computing. The MISD machine is not practical or used, it only stands in the scheme to make the classification complete. The MIMD model of execution is the most advanced where different instruction streams operate on different data sets. This is the other very important and most complex class of parallel computing systems.

Classification by address-space

We can also group parallel machines by the address space available to processors: *shared address space* versus *distributed address space*. In the shared address space model we communicate via shared variables, in the non-shared address space we use explicit message passing for data exchange.

Classification by interconnection networks

We can also classify systems by the type of interconnection network used. In *static* interconnection systems we do not use switches, processors are connected to each other by static communication links. In *dynamic* systems, the components are connected together through switches. This classification can be refined by the topology of the interconnects.

Classification by granularity

We distinguish among coarse, medium and fine grained parallelism. This refers to the amount of computation each processor does in the system. A *fine granularity* system means that processors carry out very simple calculations, typically few instructions, and consequently we need a large number of processors for good performance. In a *coarse grained system* very few processor carries out few big computational tasks. *Medium granularity* is between coarse and fine ones.

References

[8] A. L. DeCegama, *The technology of parallel processing*. Prentice-Hall International, 1989.

[9] D. C. Winsor, “BUS AND CACHE MEMORY ORGANIZATIONS FOR MULTIPROCESSORS,” the University of Michigan, 1989.

[10] S. Thakkar, M. Dubois, A. T. Laundrie, and G. S. Sohi, “Scalable Shared-Memory Multiprocessor Architectures,” *IEEE Computer*, no. June, pp. 71–74, 1990.

[11] L. Stensröm, “Reducing Contention in Shared-Memory Multiprocessors,” *IEEE Comput.*, no. November, pp. 26–37, 1988.

[12] J. Archibald and J.-L. Baer, “Cache coherence protocols: evaluation using a multiprocessor simulation model,” *ACM Trans. Comput. Syst.*, vol. 4, no. 4, pp. 273–298, Sep. 1986.

[13] H. Sutter, “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software,” *Dr. Dobb’s Journal*.

[14] L. J. FLYNN, “Intel Halts Development of 2 New Microprocessors - New York Times,” *The New York Times*, 08-May-2004.

[15] R. M. (Intel C. . Ramanathan, “Intel Multi-Core Processors Making the Move to Quad-Core and Beyond,” 2006.

[16] Intel, “An Introduction to the Intel QuickPath Interconnect.” 2009.

[17] Oracle, “Oracle’s SPARC T5-2 , SPARC T5-4, SPARC T5-8, and SPARC T5-1B Server Architecture,” 2014.

[18] M. J. Flynn, “Some Computer Organizations and Their Effectiveness,” *IEEE Trans. Comput.*, vol. C–21, no. 9, pp. 948–960, Sep. 1972.

5 MULTI-THREADED PROGRAMMING

In shared-memory parallel computers, we have access to the entire memory – global address space – within the program. Consequently, the different parts of our program that are to be executed in parallel, can communicate with one another via *shared variables*. This allows us to use the familiar sequential programming concepts for writing parallel programs. Indeed, there are many parallel programming languages that are 'only' extensions of sequential languages. Of course, there are also languages that were specifically designed for parallel programming.

All languages relying on *memory variables* for data exchange and communication among processes are called shared-memory parallel programming languages, and the programming methodology is called the shared-memory paradigm or shared-memory programming. The common theme in shared-memory programs is the notion of *thread*. The term thread comes from the expression of *single thread of control* that refers to a set of instructions executed in a sequence. These threads can be executed in shared-memory programs in parallel. A *thread*, therefore, is the *equivalent of the process* used to describe building blocks of an abstract parallel program. Shared-memory programs use threads to designate parts of a code that can be executed in parallel.

A thread is like a traditional sequential program, except that it cannot be executed on its own. A thread can only be executed by the main program it is part of. Shared-memory languages hence need to be able to manage the *life cycle of threads*; we will – as a minimum – need the ability to *create* and *destroy* threads. Since threads can be executed in parallel, there is a potential problem during program execution; threads can access and manipulate the same variable at the same time. This is a classic resource contention problem and, as we will see, can lead to non-deterministic program behaviour. This is the most crucial issue in shared-memory programming; therefore, before looking at the most popular parallel programming languages, we discuss the fundamentals of thread-based programming.

Multithreaded programs can be executed on a single processor core as well[12]. In this case, threads are not executed in parallel but concurrently. Threads compete for the processor core and only one thread executes at any one time. These systems are called concurrent systems and developing programs

[12] Think of situations where the number of threads is higher than the available processors. In this case multiple threads are scheduled for execution on one processor.

executed in this fashion is called *concurrent programming.* Since concurrent programs share the same execution problems of the shared variables and become parallel programs in the presence of multiple cores, we will discuss the fundamentals of thread-based programming within the context of concurrent programming, having in mind that everything we cover here also holds for parallel programming and program execution.

5.1 FUNDAMENTALS OF CONCURRENT PROGRAMMING

As said earlier, a parallel or concurrent program is built from processes. In the shared-memory programming world, these processes are normally implemented by threads; therefore, the words *thread* and *process* are often used interchangeably. In this book, when talking about general concepts, we will use the term *process* and only use the term *thread* when it is related to a particular language implementation.

5.1.1 Process Switching

In a concurrent program, multiple processes are executed[13] by a single CPU. Since the processor can only execute one thread of instructions, only one process will execute at any time. To advance the execution of all processes, they need to be scheduled for execution in a fair way. This, however, requires that the processor can suspend the execution of any process at any time to turn to the execution of another one.

The concept of executing a set of processes in turns is called *process switching*. Let us assume for now that processes will be executed in a time-sliced fashion. Each process has an allocated time-slice window of execution and executes its set of instructions until the time slice is over. Then the next thread starts executing its instructions for another time-slice period. Let us now look at the consequences of this seemingly simple process switching mechanism.

Process state

Every process is a simple sequential program; it has a set of instructions and a set of variables it manipulates during execution. One given set of values of the process variables is called the *internal state* of the process. It is evident that a process can have a very large number of states. In order to implement process switching correctly, we need to be able to save and restore the state of a process

[13] In concurrent programs the main reason for using threads is to improve responsiveness in user interfaces and server applications as well as to hide the effects of blocking I/O operations and long running operations.

at any point. Since the state of the process appears to the processor as the content of its registers during the execution of the given thread, process switching implies the ability to save and restore the value of the program counter and all the other registers of the processor.

The execution of the process is governed by a simple *execution state* diagram shown in Figure 5-1. By default, each process is created with an *execution state* READY. Note that the execution state is not the same as the internal state of the thread. The meaning of the READY execution state is that the process is ready for execution. The READY state is associated with a process queue (ready queue) that holds the processes waiting for execution. Only one of these processes can be in the RUNNING state, which means that the thread is currently being executed by the CPU. The transition from READY to RUNNING state is triggered by the *restore* event; this happens when the runtime system fetches the next process from the ready queue.

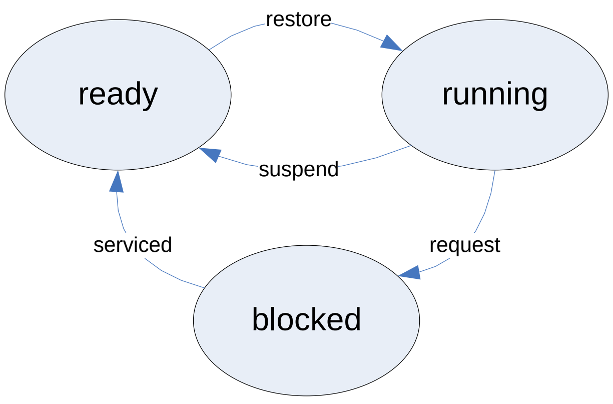


Figure 5-1 State diagram of the thread execution process.

When the time-slice of the RUNNING process is over, the process is *suspended* and the process is inserted to the end of the ready queue. Note that upon restore, the state of the process is loaded into the CPU in order to continue from the last saved state. The suspend event, similarly, automatically saves the process state. If a process performs an I/O operation that does not use the CPU, the process is put into a separate queue that holds processes blocked due to an I/O request. When the data is available, i.e. the process is serviced, the execution state of the process is changed from BLOCKED to READY and the process is removed from the blocked queue and inserted into the ready queue.

Switching processes is the responsibility of the concurrent runtime system. We are not able to control when to activate which process. This leads to the most interesting consequence of this process switching mechanism; non-determinism. In other words, we cannot guarantee the execution order of the instructions of the processes. A program is *non-deterministic* if due to potentially different instruction order in consecutive program executions it produces different results for the same inputs. This is clearly not acceptable; therefore, a concurrent program must use constructs to ensure that the program behaves in a deterministic way. As we will see, achieving deterministic behaviour is not always simple.

Assume that we have three concurrent processes P_1, P_2 and P_3 in our program. Since the processor can execute only one instruction at a time, processes will compete for the processor in order to execute their own instructions. To guarantee fairness, let us assume that the processor switches among processes after some predefined time-slice period has elapsed.

Each process consists of a sequence of instructions, which we denote as $a_1 \ldots a_n$ for P_1, $b_1 \ldots b_n$ for P_2 and $c_1 \ldots c_n$ for P_3. Further, we assume that instructions a_i, b_i and c_i are not interruptible.

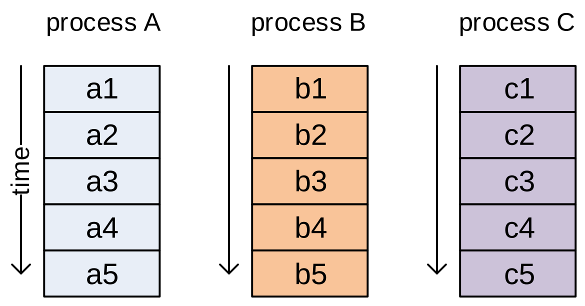
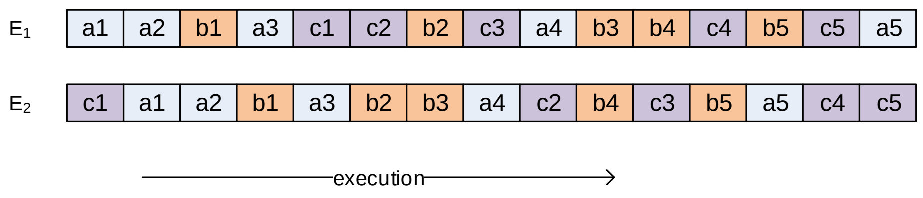


Figure 5-2 The execution order of instructions in processes A, B and C.

Since the processor cannot execute the instructions of all three processes at the same time, each process will execute for a given time then another process receives the right to execute. The execution of a process can be interrupted after any instruction and switched over to another process resulting in an execution profile E_1 (with a particular order of instructions) shown in Figure 5-3.



E1
a1 a2 b1 a3 c1 c2 b2 c3 a4 b3 b4 c4 b5 c5 a5
E2
c1 a1 a2 b1 a3 b2 b3 a4 c2 b4 c3 b5 a5 c4 c5
execution

Figure 5-3 Hypothetical interleaved execution orders of the instructions of processes A, B and C.

However, as switching from one process to another can happen at any point, the execution profile E_2 is equally likely to happen. Note that the order of instructions is different this time. Note also, that the order of instructions in a *given process* is always the same, 1,2,3,…,n. We can never have e.g. a_3 be executed before a_1 or a_2!

This form of execution, where instructions of different processes are executed in an intermixed fashion is called *interleaving*. Let us try to describe this behaviour more formally. Assume that each process executes a sequence of statements. Let us call these statements atomic actions (we explain what this means in a minute). Each process has a given state at any moment during the execution. The statements in the processes will generate state transformations within the processes. As the state of the concurrent program is the union of the individual states, each interleaved statement execution can transform the state.

One such sequence of state transformation is called the *process* (or *program*) *history*. Since many different execution orders are possible, there are several possible state transformation sequences or histories, as well. The exact number of histories for *n* processes each with a possible *m* different states is

$$K = \frac{nm!}{(m!)^n}$$

This is an enormously large number for even small values of *n* and *m*. The consequence of this is that we cannot evaluate all the possible execution histories if we want to show the correctness of our concurrent program. Concurrent programming thus is about how to ensure that execution is performed in a way that only valid state transformations (histories) are created. But to understand the problem of the incorrect state, we have to go back to the notion of atomic actions.

5.1.2 Atomic actions

We assumed earlier that process statements are atomic actions. *Atomic actions are such statements that cause indivisible state transformation*. A state transformation is indivisible if only the *initial* and *final states* are visible to other processes. There is no intermediate state, or if there is any, it is not visible to other processes, i.e. it will not affect their internal states. In plain language: if a processor starts the execution of an atomic action, it will always complete that instruction and will never switch to another process during this instruction. There are two types of atomic actions we will discuss:

- *fine-grain* atomic actions and
- *composite* (or *coarse-grain*) atomic actions.

Fine-grain atomic actions

Fine grain atomic actions are non-interruptible *hardware instructions* (processor assembly language instructions). Disabling interrupts is necessary in order to guarantee indivisible state transformations (not to suspend execution within the instruction).

High-level language statements are implemented by a sequence of these machine instructions. Even if the individual instructions are atomic, i.e. non-interruptible, the sequence is interruptible; process switching can happen between any two atomic actions. This fact has very serious consequences as we show in the following example.

We would like to execute the following program statement (assume that `x` is initialised to zero):

```
x := x + 1
```

This statement can be implemented by the following hypothetical machine instructions:

```
load x
add 1
store x
```

The execution can be interrupted after each instruction. If we stop after the `load` instruction, `x` is 0; if we stop after the `add`, the register involved has a value of 1 but it is not stored in variable `x` yet. We can only produce the correct result if we stop after the `store` instruction. It is clear that the intermediate states are clearly visible that can unfortunately influence the result of other processes!

Let us now see a more complex example involving two concurrent processes. Here we have two processes A and B executed concurrently. Process B assigns values to y and z, and process A adds these values together and assigns it to x. What is the final value of x? We would expect a value 3, but this is not the only possible result.

```
                    // initialisation

                    x = 0;
                    y = 0;
                    z = 0;

// process A                        // process B
   x = y + z;                          y = 1;
                                       z = 2;
```

To answer the question, let us list the possible interleavings (order of executed instructions) of the program and analyse what result the various sequences will produce. To make the examples easier to follow we colour-coded the processes.

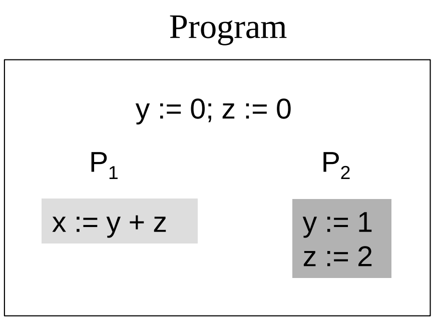


Figure 5-4 The structure of the two processe example program.

Execution history *A* in Figure 5-5 illustrates what happens if process P_1 executes first and P_2 follows. The initial values of y and z are added together and stored in x, hence x = 0. The assignments in P_2 do not have effect on P_1.

In execution history *B*, the first assignment of P$_2$ is executed first, therefore the values 1 and 0 will be added together in P$_1$. The final value of x is now 1.

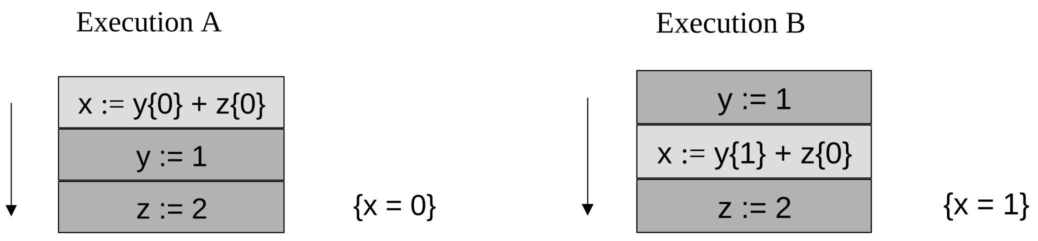


Figure 5-5 *Two different interleavings of the same program. The results are different; x=0 or x=1.*

In execution history *C* (shown in Figure 5-6), first process P$_2$ is executed completely, hence the values 1 and 2 are added together in P$_1$. The final value of x is the expected correct value, 3.

There is one more possible execution history, however, but for this we have to assume that the addition in P$_1$ is executed by the following sequence of machine instructions.

```
load y
add z
store x
```

Assume that P$_1$ executes first but it is interrupted after the `load y` instruction. The register holds 0. Now we switch to P$_2$ and it executes completely. Since z is 2, this value is added to the register and later stored in x. The final value of x is therefore 2 (!) and this is the result of an intermediate state of one process affecting the result of another process.

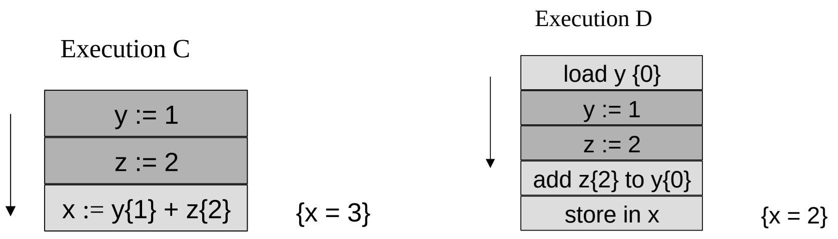


Figure 5-6 *Two further possible interleavings of the same program. The results are different; x=3 or x=2.*

We have demonstrated that depending on how the statements and instructions are interleaved, *x* can become 0, 1, 2 or 3. Next, we continue with how we can ensure our program becomes deterministic and correct.

5.1.3 Composite atomic actions

If we could eliminate the possibility of interrupting a sequence of atomic actions, we could guarantee that inconsistent intermediate states are not visible to other processes. A sequence of statements grouped together into a "non-interruptible" block is called *composite atomic action*. The composite atomic action works exactly like the instruction-level atomic action; the composite one effectively mimics the behaviour of the fine-grained atomic action.

But what part of a process should be executed as an atomic action? If the entire process is one composite atomic action, then the process will run from the beginning to the end without the possibility to switch to any other process. The efficiency of our concurrent program may decrease since executing other processes during e.g. I/O statements will be disabled. The next section will explain how to group statements into atomic actions.

5.2 The Critical Section

The *critical section* of a program code is a sequence of instructions that manipulate shared variables and hence must be executed with mutual exclusion with respect to the critical sections of other concurrent processes. *Mutual exclusion* means that when a process accesses shared variables in its critical section, no other process can have access to the same shared variables (that is, the other processes cannot execute their own critical section for this variable). This ensures that processes cannot interfere with each other; consequently the critical section will implement the composite atomic action operation.

We only need to turn those code sections atomic that manipulate shared variables. Process-local variable access can be executed in a non-atomic fashion without problems. Consequently, the execution of any concurrent process will include a combination of critical and non-critical sections of code, such as:

```
process
{
        non-critical section
        critical section
        non-critical section
        critical section
        ...
}
```

The next step in our discussion is to explore how one can turn a sequence of instructions into a critical section of code. First, we need some way of marking a code section as the 'critical part', then we must ensure it works as expected, i.e. enforcing mutual exclusion during execution.

5.2.1 The Critical Section Problem

In the following example *n* processes repeatedly execute critical and non-critical sections of code. The start and end of the critical sections are marked by a lock and unlock operation.

```
process
{
        while (true)
        {
            lock;   //marks the start of the critical section
                ...code for the critical section;
            unlock; //end of the critical section
                ...code for the non-critical section;
        }
}
```

The lock/unlock operations are not simple operations. They must meet several requirements to achieve correct behaviour. The requirements are specified as a set of rules that we will call *entry* and *exit protocols*. The protocols have to ensure that access to the critical section will be correct, that is, enforce rules as to when processes can enter or leave their critical section. The following four properties must be satisfied by the entry and exit protocols.

Mutual Exclusion – At most one process at a time is executing its critical section. Without this rule, several processes can manipulate the same shared variable at the same time.

Absence of Deadlock/Livelock – If two processes are trying to enter their critical sections, at least one will succeed. Without this rule, there may be processes that will never enter or leave the critical section thereby preventing other processes in proceeding. See definition and details in Section 5.2.2.

No Unnecessary Delay – If no other process is executing its critical section, a process will not be delayed in its entry protocol. This rule is required in order to create an efficient and fast implementation of the protocols.

Eventual Entry – A process that is trying to enter its critical section will eventually succeed. This rule ensures that all processes will be able to execute their critical section, no process will be locked out from execution.

One would think naively that we only need a language keyword for the lock/unlock operations and the problem is solved. Remember that every language keyword or instruction is implemented by a set of machine instructions. Consequently, we want to create composite atomic actions with instructions that are also non-atomic! Our central problem is hence to design implementations for the lock/unlock operations that satisfy all the four properties of the entry and exit protocols. As we will see, this is not a trivial task.

5.2.2 Solving the Critical Section Problem: first attempt

We will now look at various solutions to the critical section problem and analyse whether or not they work correctly. Initially we will consider only two-process problems in order to simplify the examples and their explanation. We will see later how they can be generalised for *n* processes.

Before describing the first solution, we need to define a few terms. A process will be called *blocked* if it is in the blocked process queue, waiting for a system request to be serviced. A process is *busy waiting*, or spinning, if it is executing a loop waiting for a condition to change. For example expressed in the C programming language as:

```
while (condition);
```

Busy waiting is not an efficient solution as it wastes CPU cycles (generates 100% CPU load).

A process is *deadlocked* if it is blocked waiting for a condition that will never become true. A process is *livelocked* if it is spinning in a loop waiting for a condition that will never become true. Livelock is the busy-waiting analogue of deadlock.

This first example uses lock() and unlock() functions to implement the entry and exit protocols. The only shared variable used is `flag` whose role is to signal that a process is executing its critical section. The example is described in C (like) language.

```
bool flag = false;

void lock()
{
    while (flag);      //wait for other process to leave CS
    flag = true;       //set flag to lock critical section
}

void unlock()
{
    flag = false;      //clear flag when exiting CS
}

process P1
{
    while (true)
    {
        lock();
        printf("P1 enters its CS");
        sleep(random amount);  //critical section
        unlock();
        printf("P1 exits its CS");
        sleep(random amount);  //non-critical section
    }
}

process P2
{
    while (true)
    {
        lock();
        printf("P2 enters its CS");
        sleep(random amount);  //critical section
        unlock();
        printf("P2 exits its CS");
        sleep(random amount);  //non-critical section
    }
}
```

The expected behaviour of this program is as follows. At start, the value of **`flag`** is `false`. The first process executing the `lock()` operation will skip the spin loop and set the **`flag`** to `true`. The other process will wait in the loop of the `lock()` operation until the first one clears the flag in the `unlock()` function. Then the second process enters the critical section and the execution continues following this pattern.

This solution seems to work correctly but there is, however, one serious error in this solution. Remember, process switching can happen at any point in a concurrent program. Let us see what happens if switching occurs within the `lock()` function. Assume that the first process reaches the `while (flag)` statement and reads `flag = false`. It will skip the loop but before executing the `flag = true` assignment, the system switches the second process, which will also start the `lock()` function and execute the `while (flag)` statement. The second process will read `flag = false` and skip the loop. Consequently, both processes will enter their critical section at the same time! *This solution therefore does not satisfy the Mutual Exclusion property of the entry protocol*.

5.2.3 Solving the Critical Section Problem: second attempt

The following solution uses two flags to indicate which of the two processes are in the critical section. This will hopefully correct the previous problem encountered with two processes reading one flag. Also, to make mutual exclusion correct, the order of the spin loop and the flag state change is swapped. This will guarantee that the two processes cannot read a 'false' value and consequently enter the critical section together.

Let us concentrate on the state of the flags and see the working mechanism of this algorithm. The lock procedure will set its *own flag* to true then check the flag of the other process and wait in a spin loop if the other is currently in the critical section.

```
bool flag1 = false;
bool flag2 = false;

void lock(bool myflag, bool itsflag)
{
myflag = true;  // intend to lock critical section
while (itsflag);  // wait for other process to finish
}

void unlock(bool myflag)
{
  myflag = false;            # allow other process access
}

process P1
{
    while (true)
    {
        lock(flag1, flag2)
```

```
            printf("P1 enters its CS");
            sleep(random amount);  //critical section
            unlock(flag1);
            printf("P1 enters its CS");
            sleep(random amount);  //non-critical section
          }
      }

      process P2
      {
          while (true)
          {
          lock(flag2, flag1)
              printf("P2 enters its CS");
              sleep(random amount);  //critical section
              unlock(flag2);
              printf("P2 enters its CS");
              sleep(random amount); //non-critical section
          }
      }
```

To analyse the operation of this solution, let us look at the state transition diagram of the program shown in Figure 5-7. One particular state of the program is specified by the combination of values of `flag1` and `flag2`. As we can see, the initial state of 00 is changed by either the `lock()` procedure in P1 or P2. For lock$_{P1}$() P1 will enter the critical section, for lock$_{P2}$() P2 will enter. The system returns to its initial state whenever P1 or P2 executes the `unlock()` procedure.

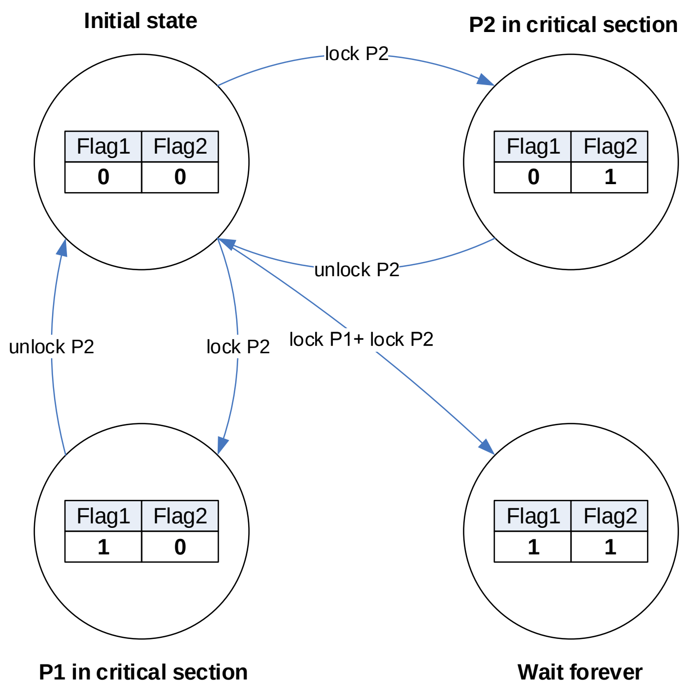


Figure 5-7 State transition diagram of the two-flag version.

If, however, P1 and P2 happens to set their flags to 'true' value at the same time – process switching occurs right after the `flag1` and `flag2` assignments – they will both read 'true' values in the spin loops and wait forever. A livelock situation will occur and since neither process can resolve this situation, *this solution fails to satisfy the Eventual Entry property of the entry protocol*.

5.2.4 Solving the Critical Section Problem: third attempt

The third solution tries to correct the previous one by attempting to resolve the livelock situation. Let us concentrate only at the case when both flags are 'true' (for the other states the solution already works correctly and this version is the same in this respect). If both flags are true, both processes will enter the loop. The first statement in the loop, however, changes the processes own flag to let the other one proceed into the critical section (as setting one flag 'false' will terminate the loop in the other process). After a given time the process sets its flag back to true, hoping that it can enter the critical section.

```
process critical2
bool flag1 = false;
bool flag2 = false;

void lock(bool myflag, bool itsflag)
{
    myflag = true;            // intend to lock critical section
```

```
        while (itsflag)         // other process has access
        {
            myflag = false;    // relinquish claim
            sleep(10);
            myflag = true;     // try again
        }
    }

    void unlock(bool myflag)
    {
        myflag = false;         // allow other process access
    }

    process P1
    {
        while (true)
        {
            lock(flag1, flag2);
            printf("P1 enters its CS");
            sleep(random amount);  //critical section
            unlock(flag1);
            printf("P1 enters its CS");
            sleep(random amount); //non-critical section
        }
    }

    process P2
    {
        while (true)
        {
            lock(flag2, flag1);
            printf("P2 enters its CS");
            sleep(random amount);  //critical section
            unlock(flag2);
            printf("P2 enters its CS");
            sleep(random amount); //non-critical section
        }
    }
```

While this solution looks very promising, we still have a little problem. Assume that in the while loop of the `lock()` function of both processes change their flag to 'false' exactly at the same time. Then they will wait the same amount of time and set their flags to true again. Then the condition test in the loop will find 'true' value for both loops – livelock again, due to this self-synchronised, lock-step style execution.

By changing the waiting time – using e.g. sleep() – to a random value in the lock() function, we can ensure that livelock will not occur, but this will have an unfortunate consequence – the solution will *not satisfy the No unnecessary Delay property*. A process cannot enter the critical section immediately; it will (unnecessarily) have to wait for the sleep to finish.

5.3 Correct Critical Section Solutions

The critical section problem was first solved by Dekker for two processes [19]. Dijkstra generalized it to *n* processes but this solution did not guarantee eventual entry [20]. The first solution to guarantee eventual entry was by D.E Knuth [21].

5.3.1 The Tie Breaker Algorithm

The Tie-Breaker was published by Peterson [22]. It is simpler than Dekker's algorithm and generalizes more easily to *n* processes. It guarantees all the four properties, including eventual entry. Peterson's original algorithm is for two processes, this is what we will discuss below. Its *n* process version was given by Hofri [x].

The problem with attempt 3 was that both `flag1` and `flag2` could be `true` at the same time and hence the processes will livelock in the while loop. In Peterson's solution, we have an extra variable in the program and few extra statements in the lock code to resolve the possible emerging livelock situation. Each process is assigned a unique ID number. In our example this is 1 for P1 and 2 for P2. They store this value in the variable '*me*'. If both flag1 and flag2 are 'true' the variable `turn` will have to resolve the conflict and decide who can enter the critical section, i.e. break the tie situation; hence the name for the algorithm.

Again, let us see the execution of the lock routine in a space-time diagram (Figure 5-7). If P1 assigns `turn` first then P2 sets its `flag2` to true, P1's while loop spins as its condition evaluates to true. While P1 is spinning, P2 will change its `turn` to the final value of 2, and as a consequence, P1's while loop will become false and so is the whole condition resulting in P1 entering the critical section. If P2 changes turn first it will work the other way around and P2 will enter.

```
program TieBreaker()
    // two-process tie-breaker algorithm

    bool flag1 = false;
    bool flag2 = false;
    int turn = 1;

    void lock(bool myflag, bool itsflag, int me)
    {
      myflag = true;             # intend to lock critical section
      turn = me;
      while (itsflag && (turn == me));
    }

    void unlock(bool myflag)
    {
      myflag = false;            # allow other process access
    }

 process P1
 {
    while (true)
    {
    lock(flag1, flag2, 1);
      printf("P1 enters its CS");
      sleep(random amount);  //critical section
      unlock(flag1);
      printf("P1 enters its CS");
      sleep(random amount);  //non-critical section
    }
 }

 process P2
 {
   while (true)
    {
    lock(flag2, flag1, 2);
      printf("P2 enters its CS");
      sleep(random amount);  //critical section
      unlock(flag2);
      printf("P2 enters its CS");
      sleep(random amount);  //non-critical section
    }
 }
```

5.3.2 Hardware Support

The main problem with all attempts and solutions we have seen so far is that they all use busy waiting. Busy waiting is a very inefficient way of creating entry protocols for critical sections. It only works well for programs in which the size of the critical section is very small. The spin loops will consume processor time even though the process is not doing any useful computation. We would like to see an implementation that does not unnecessarily use the processors.

Fortunately, modern CPUs provide certain machine instructions (primitive atomic actions) that can provide solutions to this problem. The most common such machine instructions are: *Test-and-Set* and *Fetch-and-Add*. The Test-and-Set operation *TS*(x) is defined as an indivisible sequence of the following instructions:

```
temp = x;
x = true;
return temp;
```

Hence, the precondition {x=false} will result in {x=true} and {TS(x)=false}. The precondition {x=true} will result in {x=true} and {TS(x)=true}. How can this instruction be used for our purposes? If we take a look at the first attempt to solve the critical section problem, we can see that the lock routine tests the value of flag and sets it to true once the loop terminates.

The same effect can be achieved using *TS*(x). The entry protocol of our first attempt for the critical section problem in this case will be as follows (compare this to the original in attempt 1):

```
void lock()
{
   while TS(flag);  //wait for other process to leave the critical section and
                    //set flag to lock critical section
}
```

Not only does this mean a more efficient implementation of the entry protocol, but this version is a now correct solution. Since *TS*(x) is indivisible, it is impossible for processes to read `false` value at the same time — only one process will be able to enter the critical section! The value of *x* is set to true by the first process to indicate entry to the critical section and the second process will read `true` value that it will not alter to let the first process correctly signal the end of the critical section.

The other similar instruction commonly used is the Fetch-and-Add operation *FA*(x,y), which is defined as the indivisible sequence of the following instructions:

```
temp := x;
x := x+y;
return temp;
```

This operation can be used to implement the 'Ticket' algorithm[14], which correctly solves the critical section problem for any number of processes, as well as semaphores that we will discuss in the next section. Studying the details of this algorithm is left as a homework for the more interested students.

5.4 SEMAPHORES

There are several problems with the previous solutions for the critical section problem. Firstly, there is no clear separation of variables used for critical section access control and other shared global variables used for other purposes in the program. Secondly, operations on the shared variables are dispersed throughout the entire source code making it very difficult to check the correctness of the solution or modify the program without introducing new errors. Thirdly, the busy wait method is very inefficient.

There must be some better way of achieving the same effect (a correct critical section protocol) more efficiently and creating a more understandable code. Actually, there are several such methods. One of them is the *semaphore* method that we will discuss in this section.

The semaphore in concurrent programming originates from the semaphores used in railway systems to avoid two or more trains entering a single rail track. Railways use signals at both ends of the single-track section and if one train is in the section the signal stops all other trains outside the section. To the analogy of the railway system, the Dutch computer scientist Edsger W. Dijkstra invented the semaphore concept for concurrent programs in 1968 [19]. The semaphore is a variable on which two

[14] http://research.microsoft.com/en-us/um/people/lamport/pubs/bakery.pdf

operations are defined: *set* and *clear*. Originally, these operations were labelled as *P* and *V*. *P* stands for the Dutch word *Probeer* (try), the letter *V* for *Verhoog* (increment) as suggested by Dijkstra[15].

Let *sem* be a semaphore type variable and we define two atomic operations *P* and *V* on it as:

- *P*(*sem*) to set,
- *V*(*sem*) to clear the *sem* semaphore variable.

Let us see now how this can be used for solving the critical section problem. Again, just as at the railways, we would like to protect the critical section at both ends with semaphores and we know that one has to be set and the other end has to be cleared. In pseudo code form, the critical section is marked the following way:

```
P(sem)
   critical section
V(sem)
   non-critical section
```

This is the same for any process in the system with a critical section in it. How will this work? It is easy to see that if a process is in its non-critical section then it executed one *P* operation and one *V* operation. Their difference is zero. If all processes except one are in their non-critical section and one is in its critical section, there is one more *P* operations executed than *V* (the one in the critical section is yet to execute the *V* operation). Hence, the difference of P and V is greater than zero. From this we can derive the following.

Assume an integer semaphor *S* with initial value 0. Let *nP* denote the number of completed *P* operations and *nV* the number of completed *V* operations. Then, a process can enter the critical section if and only if there is no other process in the critical section, i.e. all processes are currently outside the critical section. This is equivalent to

$$nP = nV,$$

since the number of completed *P* and *V* operations must be equal. Semaphores are normally implemented with a process queue. If the semaphore value is 0, then the *P* operation will add the process to the process queue. When the semaphore count is increased and there are processes in the process queue, one will be fetched from the queue and that can enter its critical section. The operation

[15] E. Dijsktra, On semaphores (in Dutch), http://www.cs.utexas.edu/~EWD/ewd00xx/EWD74.PDF

of the integer semaphore operations are defined as follows. *P* will always decrement *S* by one and *V* will increment *S* by one. More formally, the operations are given by the following code segment.

```
P(s)
if (s > 0)
        s = s - 1;
else if (s = 0)
{
        process becomes blocked
        add process to semaphore queue
}

V(s)
if (queue = empty)
        s = 1;
else
{
        remove head process from queue
        reschedule on ready queue
}
```

If the semaphore is initialised to a value greater than 1, we call it a *general semaphore*. Such a semaphore will not implement mutual exclusion but will allow access to a several resources. The semaphore will allow access to the resources until the number reaches zero, in which case processes will have to wait until resources become available again.

If the semaphore is initialised to 1, it will only have 1 and 0 values. This type of semaphore is called *binary semaphore*, which is used to implement mutual exclusion in critical sections, since only one process can execute the critical section, all the other will have to wait for their turn. Binary semaphores are similar to the *mutex* construct in operating systems. Normally, semaphores have very efficient implementations; they rely on hardware instructions and operating system support as well.

5.4.1 Case Study: the Dining Philosophers

We illustrate the use of general semaphores with a classic concurrent programming problem, the Dining Philosophers [23].

Five (or more generally, *N*) philosophers sit around a circular table. Each philosopher spends his life alternately eating and thinking. In the centre of the table is a large plate of spaghetti. Because the spaghetti is long and tangled, a philosopher must use two forks to get a helping. Unfortunately, the table is set with only one fork between each pair of philosophers and each philosopher can use only the forks to his immediate left and right.

The problem is to write a program to simulate the behaviour of the philosophers that can eat and think to an indefinite time. The program must avoid the situation in which each philosopher picks up exactly *one* fork and all philosophers starve to death.

Let us first analyse the problem. It is clear that the critical resource is the fork. They do not have enough forks, and since two or more philosophers cannot eat with the same fork at the same time, only one philosopher can use a fork at any time. If a philosopher needs two forks to eat, at most 2 philosophers can eat at the same time (2 philosophers x 2 forks → 4 forks out of the 5 are in use).

To find a solution we should try to map our problem to a critical section problem. The fork is a critical section as only one process (philosopher) can access it at a time. We have, however, several forks, therefore more than one critical section.

The simplest approach to the problem is to have an array of semaphores, one array element for each fork. In pseudo code:

```
Semaphore forks[5];
```

This way we can signal whether or not a particular fork is in use. Then the corresponding process (the philosopher that can pick up that fork) will be able to detect whether it is possible to pick up the fork.

If we number the philosophers from 1 to 5, and know that each of them can only use the fork to the immediate left or right, each of them will pick up `fork[i]` and `fork[i+1]`. Not quite so. Philosopher 5 cannot pick up fork 5 and 6 since we only have 5 forks. We have to use a different numbering scheme. Since the philosophers sit around the table, philosopher 5 has philosopher 4 and 1 as neighbours. The same is true for the forks, so philosopher 5 should pick up fork 5 and 1. The correct numbering can be achieved using modulo arithmetic. Each philosopher should pick up `fork[i]` and `fork[i mod 5 + 1]`.

Using this, the next program illustrates our first attempt to solve the problem. We shall now see how the program works and find out whether or not this is a correct solution.

```
DiningPhilosophers()
   // Dining Philosophers

     int N;
     N = 5; // number of philosophers
     semaphore fork[N];
     initialise fork[i] to 1;

     process Phil
     {
        i is process ID
        while (true)
        {
          P(fork[i]); sleep(1); P(fork[i % N + 1]);
          printf("Philosopher %d is eating", i)     // eat
          sleep(random amount);                     // eating time
          V(fork[i]); V(fork[i % N + 1])
          printf("Philosopher %d is thinking", i)   // think
          sleep(random amount);                     // thinking time
        }
     }
end DiningPhilosophers
```

The problem with this algorithm is that we can have a process switch between the two P operations at the beginning of the loop, resulting in a situation where each philosopher picks up the first fork but none of them will be able to pick up the second one. They will starve to death! In other words, the algorithm will deadlock.

This situation arises because each philosopher first picks up the fork to the left. If we prevent at least one of them from this situation, by making him to pick up the right fork first, circular waiting will not occur. Four philosophers will be able to pick up a fork and the fifth one will have none. Then two from the four with one fork will be able to pick up the second fork. Finally, this is a correct solution to the problem.

```
DiningPhilosophers()
```

```
// Dining Philosophers

  int N;
  N = 5; // number of philosophers
  semaphore fork[N];
  initialise fork[i] to 1;

  process Phil i=0… (N-2)
  {
     i is process ID
     while (true)
     {
       P(fork[i]); sleep(1); P(fork[i % N + 1]);
       printf("Philosopher %d is eating", i)     // eat
       sleep(random amount);                     // eating time
       V(fork[i]); V(fork[i % N + 1])
       printf("Philosopher %d is thinking", i)   // think
       sleep(random amount);                     // thinking time
     }
  }
  process Phil i=N-1
  {
     i is process ID
     while (true)
     {
       P(fork[0]); sleep(1); P(fork[N - 1]);
       printf("Philosopher %d is eating", i)     // eat
       sleep(random amount);                     // eating time
       V(fork[0]); V(fork[N - 1])
       printf("Philosopher %d is thinking", i)   // think
       sleep(random amount);                     // thinking time
     }
  }

end DiningPhilosophers
```

5.4.2 Case Study: the Producer-Consumer problem

In the next classic problem, we will look at the use of the binary semaphores to provide condition synchronisation capabilities to processes. We have looked at how semaphores can be used to solve the critical section problem and achieve mutual exclusion during execution. There are situations, however, when mutual exclusion is not sufficient for the correct behaviour. In certain cases, there are dependencies between processes; one has to wait until the other finishes or produces some result.

The Producer-Consumer problem is a classic illustration of this case. Two processes, a Producer and a Consumer, communicate via a shared bounded buffer. The producer generates numbers in increasing order and sends it to the Consumer via the buffer. The Consumer reads the values from the buffer and displays them on the screen. We need to ensure that our solution will preserve the order and content of the number sequence sent from one process to the other; i.e. data cannot be lost, received multiple times or in an out-of-order fashion.

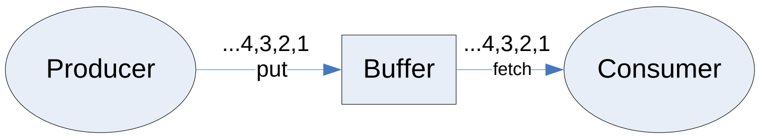


Figure 5-8 The block diagram interactions in the Producer-Consumer problem.

Since the buffer is a shared resource, access to the buffer is the critical section in both processes, consequently, we will need to implement mutual exclusion. But is that enough for the correct solution? Unfortunately, no. Mutual exclusion will only guarantee that the two processes will not read/write the buffer at the same time. Unfortunately, they can execute at very different speeds, which can result in situations such as the Producer sending values much faster than the Consumer can fetch them, hence data can be accidentally overwritten by the Producer. If the Consumer is much faster, it can read the same data value several times as new one is not yet produced.

It is clear, that we have a dependency in the operation of the two processes; data produced by the Producer must be consumed before the next value is sent. If we look at the problem from the buffer's point of view, we have to ensure that the consumer reads the buffer only when the data is ready and the producer will not overwrite the buffer if it is not empty (it will only write to an empty buffer). More formally, we can set up the following conditions:

Condition 1: *producer must wait until buffer is empty*

Condition 2: *consumer must wait until buffer is full*

One semaphore is not enough for the solution of this problem since that would only guarantee mutual exclusion, but it would not guarantee the correct order of events (reading from and writing to the buffer at the correct time). We should use two semaphores, one to control the producer and one to control the consumer.

It is time to introduce another concept at this point, the *split-binary semaphore*. Split-binary semaphores are defined by the following constraint:

$$0 \leq S_1 + S_2 + S_3 + \ldots + S_n \leq 1$$

They are very similar to binary semaphores (remember $0 \leq S \leq 1$) except that in this case we deal with more than one semaphores. Since the constraint is the same as for the binary semaphore, we can look at the split-binary semaphore as a binary semaphore cut (split) into many little binary semaphores S_i. It is clear from the definition that only one of the *n* elements can have the value 1 at any time.

In the Producer/Consumer problem, we will use a split-binary semaphore created from two binary semaphores signalling the state of the bounded buffer.

```
semaphore: empty

semaphore: full
```

The initial value of 'empty' will be 1 so the Producer is able to start the program and write into the buffer. The initial value of 'full' is 0 to make the Consumer to wait for data. Let us see the program implemented with semaphores.

```
ProducerConsumer
{
     // Multiple Producer Consumer problem with single shared
     // buffer buff
     // Solution uses split binary semaphores

     int buff;
     semaphore empty = 1;
     semaphore full = 0;  // binary semaphores

   process Producer
   {
        while (true)
        {
          // generate data x
          P(empty);             // wait until buffer empty
          buff = x;
          V(full);              // signal full
       }
    {

    process Consumer
     {
        while (true)
        {
           P(full);              // wait until buffer full
           y = buff;
           V(empty);             // signal empty
          // consume data y
       }
    {
}
```

Since the semaphore empty is 1, the P(empty) operation will succeed and producer can write the value into the buffer. The consumer will initially wait as the value of full is 0, hence the consumer process is put into the waiting queue. When the producer finishes the buffer update, it clears the full semaphore (increments its counter), signalling the consumer that is waiting at the P(full). Since the empty semaphore is not cleared yet, the producer is put on hold when trying to put the next piece of data into the buffer. As the consumer retrieves the previous value, it clears the empty semaphore that, in turn, will signal the producer to proceed. Using two semaphores in this way, the two processes run concurrently but still in the correct lock-step fashion. This example illustrated the meaning of condition synchronisation and the use of split-binary semaphores for correctly implementing condition synchronisation.

5.5 The Monitor

The semaphore is a cleaner and more efficient solution to the critical section problem than busy waiting. Yet, there are inherent problems in its use. The most important and dangerous is related to software engineering. A concurrent program can have any number of critical sections. For large software systems, with several ten or hundred thousand lines of code, the P, V operation pairs can plague the entire program. Potentially every line can become a critical section. If programmers make any mistake in such a program, e.g. accidentally deleting a V operation, or inserting a P, the program will become incorrect and behave unpredictably. Unfortunately, these concurrency errors are often very difficult to detect and correct as they depend on how the processes are interleaved. For large teams of programmers, distributing development tasks can be a nightmare if concurrent constructs intersperse the code entirely.

Several researchers (C.A.R. Hoare [24], Per Brinch Hansen [25]) came up independently with the idea of encapsulating the shared variables and the operations defined on them into a modular construct that they called monitor or secretary. The term *monitor* coined by Hoare became widely accepted and this is how now we know this very important concurrent software abstraction.

The monitor is a structure that collects critical sections into a single module. Processes using monitors no longer have critical sections, they are turned into procedure calls on the monitor. The monitor satisfies very important properties:

1. The monitor provides implicit mutual exclusion; monitor procedures are always executed one at a time.
2. Processes not gaining access to the monitor will be automatically placed into a wait queue of the monitor.
3. The monitor provides constructs for condition synchronisation that the programmer can use to signal processes about certain events and conditions.

It is important that variables of a monitor cannot be accessed directly from outside the monitor; shared data should be encapsulated in the monitor and can only be accessed by monitor procedures. A typical monitor language construct can take the following form [24]:

```
monitorname: monitor
    begin…  declarations  of data local  to  the monitor;
        procedure procname (…  formal parameters…) ;
            begin…  procedure  body..,  end;
        …declarations  of  other procedures  local  to the monitor;
        …initialization  of  local data  of the  monitor...
    end;
```

Within the program, procedures can be invoked as *monitorname.procname(parameters)*. Naturally, one can declare and use any number of monitors as required. The introduction of this construct is very important in the development of concurrent and parallel programming as the modularity it introduced was indispensable for large-scale concurrent programs. Using monitors, once can safely separate functional and concurrent code sections that i) help in using the right experts for the given task and ii) helps in testing and localising concurrency related errors.

Monitors can also be used for condition synchronisation but this has to be used by the programmer in an explicit fashion, as the monitor cannot figure out on what condition we would like to synchronise processes and how. Condition synchronisation is supported by a new variable type called *condition*. This type has a queue associated with it and two operations, *wait* and *signal*. If a process needs to wait for a condition to hold, it can enter a waiting state by calling *wait(condition_variable)*. The *wait* can only be called within a monitor, hence this has to be embedded in the procedures within which the monitor is accessed. Once the *wait* executes, the calling process is suspended and put into the queue associated with the given condition variable. When the condition the process is waiting for becomes true, the waiting process is woken up by sending it a signal via a call to *signal(conditional_variable)*. The use of condition variables is illustrated in the monitor implementation version of the Producer-Consumer algorithm.

```
(* code for Producer *)
loop
    data:= produce item
    buffer.put(data);
end loop
```

```
(* code for Consumer *)
loop
    data:= buffer.take(data);
    process data
end loop
```

```
buffer: monitor
    begin
        full, empty: condition variables
        counter: integer;
        data: integer;
            procedure put (item: integer) ;
                begin
                    if (counter > 0) then wait(full);
                    data := item;
                    counter := counter + 1;
                    signal(empty);
                end;


            function take: integer;
                begin
                    if (counter = 0) then wait(empty);
                    counter := counter - 1;
                    signal(full);
                    return data;
                end;

            (* initialization  of  local data  of the  monitor *)
            counter:= 0;
    end;
```

5.6 MULTITHREADED PROGRAMMING IN JAVA

Java is a popular object-oriented language used by millions of developers over the world, which supports the development of multi-threaded programs since the first version of the language. In this section, we look at how Java threads can be created, managed and executed in a safe and controlled manner. Since the complete description of the language is beyond the scope of this book, we assume the reader has a basic working knowledge of Java, and we only concentrate on the thread-related aspects of Java. Readers not using the Java programming language at all can safely skip this section.

5.6.1 The Java thread model

Java is a 100 % object-oriented language. Not surprisingly, threads appear in a Java program as objects. In the Java thread model, all thread objects must implement the java.lang.Runnable interface. The interface defines a no argument method run() that classes implementing the interface must provide implementation for.

```
public interface Runnable {
        public void run();
}
```

As a convenience class, the class java.lang.Thread is available that implements the Runnable interface, provides an empty-body default run() method, and also provides additional life-cycle and utility methods.

Threads are executed by the Java Virtual Machine (JVM) in a time-sliced fashion using round-robin scheduling. Threads are taking their turn and executed one after the other in a cyclic pattern. Context switching is performed automatically by the JVM, the user does not have control over it. Java threads can have different priority levels. The default priority, Normal, is represented by the Thread.NORM_PRIORITY (=5) constant. Acceptable priority levels must fall into the range of Thread.MIN_PRIORITY (=1) and Thread.MAX_PRIORITY (=10). Higher priority threads are scheduled preemptively, that is, the higher priority thread will get preference over lower priority ones. For simplicity, we assume all threads in this book as normal priority threads.

5.6.2 Creating threads

Whether we choose to implement the Runnable interface or inherit from the Thread class, we need to create a specialised class that will contain the implementation for the task the thread should perform.

We start with a simple example that creates a class named WorkerThread extending the class Thread. The code for the custom thread should be placed into the run() method in class WorkerThread.

```
public class WorkerThread extends Thread{

        private int limit;

        public WorkerThread(int n){
                limit = n;
        }

        public void run(){
                System.out.println(getName() + "running...");
                for(int i=0; i < limit; i++){
                        double y = Math.sin((double)i);
                        System.out.println("value = " + y);
                }
                System.out.println(getName() + "stopped...");
        }
}
```

Note that the run() method does not take arguments and never returns a value. The only way to pass parameters to the run() method is via instance variables set either using the constructor or special setter methods. The result of the thread execution (if any) can be retrieved with getter methods once the thread finished the execution. The example thread WorkerThread will print out values of the $f(x) = sin(x)$ function.

If we choose to implement the Runnable interface, we need few extra steps in the implementation. In addition to implementing the run() method:

```
public class WorkerThread implements Runnable{
        public void run(){
                // code from above
        }
}
```

we need to make sure the thread can be started properly. Note that this class does not have a start(). In the code section where we want to start our thread, use the following construct:

```
new Thread(new WorkerThread()).start();
```

Java threads are started by the virtual machine (JVM). From the Java program we call the `start()` method of the base Thread class to request the virtual machine to create a JVM thread and associate it with the new thread instance; in our case the WorkerThread instance. Once the JVM thread is created, the virtual machine will call the `run()` method of our thread to start the execution. Never call the `run()` method directly on a thread! That will simply perform a traditional blocking method call.

The thread will automatically stop once the `run()` method ends. In our sample program, we create two threads, t1 and t2 and start them to execute in parallel. The main method will not terminate until the threads finish.

```
public class ThreadExample {

        public static void main(String[] args){
                // instantiate class WorkerThread
                WorkerThread t1 = new WorkerThread(5);
                WorkerThread t2 = new WorkerThread(5);
                // start the threads
                t1.start();
                t2.start();
                // program exits when both threads have finished
        }
}
```

If we need to explicitly wait for the execution of some threads, for either getting their results or synchronising the start of the next processing step, we can use the `join()` method on the thread. Calling the join will block the calling thread until the thread is running. E.g., continuing with the above example:

```
        ...
        // start the threads
        t1.start();
        t2.start();
        ...
        t1.join();
        t2.join();
        // threads have finished, we can continue
```

5.6.3 Stopping the execution of a thread

Java threads do not have a `stop()` method (actually they have but it is deprecated and should never be used!). The correct way to stop a thread is to request it to stop by calling the `interrupt()` method on the thread. A well-behaved thread will respond to the interrupt request and stop any long running I/O or computational tasks. Since this is not the core topic of our discussion, we leave the details for tutorials the reader can easily consult.

5.6.4 Communication between threads

If we use threads for concurrent or parallel programs and not for independent work (such as background tasks in GUIs or client tasks in server applications) Java threads might need to communicate during the computation. This can only be performed via shared variables. Synchronisation is necessary for correct behaviour as we have seen earlier. Due to its object-oriented nature, Java uses the *monitor* concept discussed in Section 5.5. The only difference is that while in the original monitor concept, the monitor is a special data type in the language, in Java, every object can become a monitor. Each Java object – an instance of the base class Object – contains a lock that can be used to implement monitor behaviour on an object. Since every Object can only potentially become a monitor, the Object class does not provide implicit mutual exclusion. Both this and condition synchronisation has to be specified explicitly by the programmer using special keywords and/or Object methods.

5.6.5 Implementing Mutual Exclusion

The Java language keyword `synchronized` is used to mark a method or a block of code to be executed with mutual exclusion. If we execute methods with mutual exclusion, we talk about *synchronized methods*, if – however – we execute a code block with mutual exclusion, we talk about *synchronized statements*. If used at the method level, its declaration syntax is the following:

```
public synchronized returnType methodName(paramType parameter){
    ...
}
```

The keyword *synchronized* ensures that when this method is called, the lock of the object is *acquired*. If the lock is free, the method will succeed in acquiring the lock and will own it until the synchronized method finishes, when the lock is automatically *released*. If the lock is not free, it is owned by another thread, the thread attempting to acquire the lock will suspend its operation and will wait for the lock to be released.

This approach will provide synchronisation at method level granularity, which in many cases is too large. For protecting smaller code sections, we can use the *synchronized statements* approach whose syntax and use is as follows:

```
public void methodName(String param) {
    ...
    statements
    ...
    synchronized(this) {
        variable = param;
        ...
        other protected statements
    }
    ...
    statements
    ...
}
```

5.6.6 Using condition synchronisation

Condition synchronisation is similar in concept to the one we have seen in the monitors except that in Java there is no condition variable data type. The base Object class can implement condition synchronisation relying on the lock of the object discussed above. Any thread wishing to suspend its operation can invoke the `wait()` method of the base Object and signal the thread to wake up on a condition occurrence by invoking the `notify()` method. Method `wait()` is equivalent to WAIT of the monitor, `notify()` is equivalent – in principle – to SIGNAL of the monitor. The main difference is that in Java one monitor can only have one (implicit) condition variable, the lock, and different threads waiting for different conditions to hold will all wait on the same lock! It is important to note that the `wait()` method will block the caller; it will not return and continue executing other statements until the thread is woken up by the a call to the `notify()` method.

The Java Virtual Machine specification guarantees that as a result of `notify()`, **one** of the waiting threads will be woken up. The selection will happen without guarantees, most likely it will be random. Since there is no guarantee that the thread actually waiting for the particular condition will wake up, instead of the method `notify()`, we prefer the use of the method `notifyAll()`, which wakes up all waiting threads. It is then a simple matter to re-check the condition and verify that the thread was the one waiting for this condition or not. If the signal was not intended for a particular thread it will call wait to be suspended again. For this reason, besides the rule that `wait()` and `notify()`, `notifyAll()` must be called from within a synchronised method or code block, is is also important that the wait should be called within a loop that checks conditions again upon returning from the `wait()` method.

5.6.7 Example: The Producer-Consumer problem in Java

We will re-implement the Producer-Consumer problem discussed in Section 5.5 in Java to (i) show how to create and use monitor objects in Java, and (ii) illustrate the modularity of our program and the separation of concurrent and application functionalities in the code.

The main application class ProducerConsumer will start in the `main()` method which creates an instance of the bounded buffer and the Producer and Consumer threads *p* and *c*. Both threads receive a reference to the buffer via their constructor and finally the threads are started with the `start()` method. Note that the details of the thread implementations and communications are entirely hidden; in fact, by reading the code we cannot even notice that we are working with a multi-threaded program (any object can have a `start()` method).

```
public class ProducerConsumer{

    public static void main(String[] args){
        // create monitor object
        Buffer buffer = new Buffer();

        // create thread objects
        Producer p = new Producer("Producer", buffer);
        Consumer c = new Consumer("Consumer", buffer);

        // start threads
        p.start();
        c.start();
    }
}
```

Let us now turn to our thread implementations and see the producer process first. The producer stores the reference to the buffer object in a private field *buffer*, and uses it in the `run()` method to put numbers into the buffer using the `put()` method of the buffer. Note again, the lack of concurrent constructs in the code. The synchronisation required for this implementation is hidden in the monitor – buffer – object. This is a nice example of the 'separation of concern' principle. Developers responsible for application functionality can concentrate on their part without needing to worry about the monitor implementation that can be delegated to a concurrent programming expert.

```
public class Producer extends Thread{

    private Buffer buffer;

    public Producer(String name, Buffer b){
        super(name);
        buffer = b;
    }

    public void run(){
        // our special thread code
        for(int i=0; i<10; i++){
            System.out.println(getName() + ": writing: " + (i+1) );
            buffer.put(i+1);
        }
    }
}
```

The consumer thread implementation follows the same design as the producer, only the communication direction is opposite. Instead of putting, the consumer will remove data from the buffer using the buffer's `get()` method. For simplicity reasons, both processes use a hard-coded loop size of 10.

```
public class Consumer extends Thread{

    private Buffer buffer;
    public Consumer(String name, Buffer b){
        super(name);
        buffer = b;
    }

    public void run(){
        // our special thread code
        for(int i=0; i<10; i++){
            int value = buffer.get();
            System.out.println("    " + getName() +
                ": reading: " + value );
        }
    }
}
```

The most interesting part of our implementation is the buffer class that implements the monitor concept and turns our application thread-safe. We create a single-slot buffer (able to store only one data item) for simplicity reasons. The flag `hasData` will signal whether or not the buffer is full. Two methods are required on the buffer, `put()` to insert data into and `get()` to fetch data from the buffer.

Since the put and get cannot be executed at the same time, we will need synchronized methods to make them execute with mutual exclusion. Condition synchronisation will guarantee that if the producer finds a full buffer, it will wait until it becomes empty. When the consumer removes the data from the buffer, it clears the flag and wakes up the waiting threads with `notifyAll()`.

```
public class Buffer{

    private int value;
    private boolean hasData = false;

    public synchronized void put(int v){
        while (hasData){
            try{
                wait();
            }catch(InterruptedException e){
                e.printStackTrace();
            }
        }
        value = v;
        hasData = true;
        notifyAll();
    }

    public synchronized int get(){
        while (! hasData){
            try{
                wait();
            }catch(InterruptedException e){
                e.printStackTrace();
            }
        }
        hasData = false;
        notifyAll();
        return value;
    }
}
```

The output from the program is a follows. Modify the above example to see the behaviour without condition synchronisation and/or mutual exclusion.

```
Program started...
Program finished...
Producer: writing: 1
Producer: writing: 2
Producer: writing: 3
   Consumer: reading: 1
Producer: writing: 4
   Consumer: reading: 2
Producer: writing: 5
   Consumer: reading: 3
Producer: writing: 6
   Consumer: reading: 4
Producer: writing: 7
   Consumer: reading: 5
Producer: writing: 8
   Consumer: reading: 6
Producer: writing: 9
```

```
   Consumer: reading: 7
Producer: writing: 10
   Consumer: reading: 8
   Consumer: reading: 9
   Consumer: reading: 10
```

The most complicated part of this algorithm is the Buffer. Since these type of problems are common in concurrent/parallel programs, Java offers utility classes to simplify program development. The `java.util.concurrent.ArrayBlockingQueue` class can be used as a replacement for the above Buffer class. The put() and take() operations will block if the buffer is full or empty, respectively. In addition, this class can hold a configurable number of elements.

Similarly, for executing threads in a controlled manner, e.g. controlling the number of threads not to exceed a limit or to run fork-join type of problems, we can use implementations of the `java.util.concurrent.Executor` interface. These classes implement very useful execution policies that can be used effectively in parallel programs or server implementations alike.

5.6.8 Parallel example: matrix-vector multiplication

We close the Java thread programming section with a classic data-parallel example. We will multiply an *n* x *n* matrix ***A*** with an *n* x 1 vector ***x*** and generate the results in vector ***y***. The sequential algorithm is based on the following formula for calculating one element of ***y***:

$$y_i = \sum_{j=0}^{n-1} A_{ij} \cdot x_j$$

The sequential implementation is straightforward, we compute the result in a double nested for loop.

```
// declare and initialise A, x, y, size
for (int i = 0; i < size; i++) {
     double sum = 0;
     for (int j = 0; j < size; j++) {
             sum += A[i][j] * x[j];
     }
     y[i] = sum;
}
```

We can see that the complexity of this algorithm is $O(n^2)$. We execute n^2 iterations where each iteration performs 2 operations, one multiplication and one addition. Since we do not have many operations, to execute in parallel, we look for partitioning opportunities in the data. This is a classic data parallel algorithm. Each element of ***y*** can be computed independently from the others. For this, each process needs one row of matrix ***A*** and the entire vector ***x***.

$$\overbrace{\begin{bmatrix} & \cdots & \\ \vdots & \ddots & \vdots \\ & \cdots & \end{bmatrix}}^{A} \cdot \overbrace{\begin{bmatrix} \vdots \end{bmatrix}}^{x} = \overbrace{\begin{bmatrix} \vdots \end{bmatrix}}^{y}$$

If *n* is large, we cannot have each element of ***y*** computed by a processor; instead, we will partition ***A*** and ***y*** horizontally to segments or data chunks of size *n*/*p* rows, where *p* is the number of processors. Each process (thread) will have to have access to the matrix and the vectors and, in addition, will have to know where its data segment is located in the original matrix ***A*** and vector ***y***. The sequential multiplication code will be turned into a parallel one by defining a thread that will work on one segment of the data. Note the row index of ***A*** and ***y*** in the `run()` method.

```java
class MatrixVectorThread extends Thread{

      private int threads;
      private int index;
      private double[][] A;
      private double[] x;
      private double[] y;
      private int rows;

      private MatrixVectorThread(int threads, int i,
                  double[][] A, double[] x, double[] y) {
            this.threads = threads;
            this.index = i;
            this.A = A;
            this.x = x;
            this.y = y;
            this.rows = size/threads;
      }
```

```
        public void run(){
              for (int i = 0; i < rows; i++) {
                    double sum = 0;
                    for (int j = 0; j < size; j++) {
                          sum += A[index*rows+i][j] * x[j];
                    }
                    y[index*rows+i] = sum;
              }
        }
  }
```

The program will execute following the fork-join pattern; it will create the specified number of threads (set in threads), start the threads and wait for completion. Try to implement this program and measure the execution time for different problem sizes and number of threads. Plot the execution time and compute the achieved speedup!

```
// declare and initialise A, x, y, size
MatrixVectorThread[] t = new MatrixVectorThread[threads];
for (int i = 0; i < threads; i++) {
      t[i] = new MatrixVectorThread(threads, i, A, x, y);
 }
 long start = System.nanoTime(); // start measuring execution time
 for (int i = 0; i < threads; i++) {
     t[i].start();
 }
 for (int i = 0; i < threads; i++) {
     try {
         t[i].join();
     } catch (InterruptedException ex) {
      // handle exception
     }
 }
 long stop = System.nanoTime(); // stop the timer
 System.out.println("time: "+size+" "+(stop-start)/1000.0+" usec" );
```

5.7 OpenMP

The OpenMP[16] standard was created in 1997 with the support of Compaq, Digital, Intel, IBM and Silicon Graphics, in order to address the problems the diversity of shared-memory parallel computers caused as well as to create a programming model that makes it easier for programmers to create parallel programs for shared-memory computers. OpenMP not only does provide an easier, simpler entry to the field of parallel programming than other approaches, but being a *de facto* standard, it also supports portability of programs among shared-memory computers.

OpenMP was created for shared-memory parallel programming and its central notion is the thread and the fork-join style parallelism. OpenMP is not a programming language; it is an extension to existing languages (C/C++ and Fortran) to make them usable for parallel program development. The central concept of OpenMP is that it does not request the programmer to write all the parallel 'boiler-plate' code, but to indicate where parallelism is in the program and leave the actual parallel program code generation (e.g. creating and starting threads, etc) to the compiler. As we will see, this results in a nice, easy-to-understand program structure, especially useful for numerical algorithms.

Today, OpenMP is supported by most current compilers. If in doubt, consult the manuals of your favourite compiler for its OpenMP support. OpenMP is evolving as the field is maturing and new requirements appear. At the time of writing, the current version is v4.0. Note that most compilers still only support version 2.0 or 2.5.

OpenMP has APIs for the C/C++ and FORTRAN languages. In this book we only concentrate on the C version of OpenMP. In the case of C, OpenMP relies on ***compiler directives*** (for FORTRAN programs, comments are used). The advantage of using directives is that if a compiler does not support OpenMP, it will simply ignore the OpenMP directives and generate straightforward sequential programs without any program modification. If, however, the program is compiled with an OpenMP-enabled compiler, it will generate parallel code. Make sure you understand, that the compiler does NOT auto-parallelise the sequential code; it will only generate parallel code based on the instructions the programmer inserts into the code!

OpenMP also has a runtime environment and provides ***library routines*** for interfacing with the runtime and to perform synchronisation and similar operations. We will look at these functions later

[16] OpenMP homepage: http://openmp.org/wp/

in this section. There are also a set of ***environment variables*** that one can use to control/change the execution of a parallel OpenMP program without recompilation.

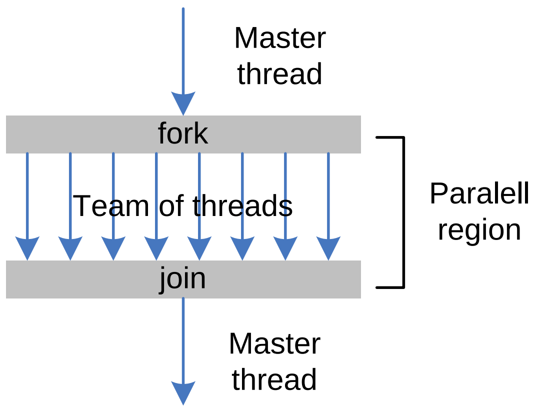
In brief, openMP provides the users with the ability to

- create a team of threads,
- share work among threads,
- synchronise threads,
- use shared and private variables, and
- control the scheduling policy used in thread execution.

In the rest of this section, we will discuss the key features of the OpenMP API and their use in more detail. Let us start with a very simple example to demonstrate the structure of an OpenMP program.

```
#pragma omp parallel if (n > threshold) \ shared(n,x,y) private(i)
{
       #pragma omp for
       for (i=0; i<n; i++)
              x[i] += y[i];
} /*-- End of parallel region --*/
```

This example performs a parallel vector addition. Besides the traditional for loop for the addition, it contains two directives. The first directive – `#pragma omp parallel` – instructs the compiler to create *n* threads, but only if *n* is greater than a predefined threshold value. The second directive – `#pragma omp for` – specifies that the loop iterations should be divided among the threads and executed in parallel. The execution pattern is shown in the following figure. The master thread creates the threads and forms what is called in OpenMP a *parallel region*. The threads in this region execute at the same time and will add different parts of the two vectors together. Once the additions are done, threads stop and the program continues in a single thread.



Do not worry if this is too much to grasp; the only important thing to realise here is that we did not have to write code for creating, starting, stopping threads, distributing data among threads, etc. In OpenMP, this is performed behind the scenes.

Let us start the detailed discussion of OpenMP with a look at the syntax and formal definition of the OpenMP constructs:

#pragma omp *directive-name [clause[[,] clause]...] new-line*

Each construct is specified with the OpenMP pragma **#pragma omp**. This is followed by a directive name and a set of directive-dependent clauses that can modify the behaviour of the directive. The most important directive name keywords are the followings:

parallel	defines a parallel *region* and creates a team of threads
for	executes iterations of a loop by the team of threads of parallel region
section	specifies structured blocks that will be distributed and executed in parallel
single	specified that the structured block should be executed by only one thread
task	specifies a specific task for execution
master	specifies a structured block to be executed by the master thread of the team of threads.
critical	ensures that the structured block is executed by a single thread at a time
barrier	specifies an explicit barrier synchronisation operation at the point of the construct in the code

5.7.1 Creating teams of threads

The most important directive in OpenMP is the directive *parallel*. This directive is used to specify a structure block (a section of code) to be executed in parallel. If the parallel directive is never used in an OpenMP program, it will be executed serially as a sequential program! The exact syntax of the parallel construct is as follows:

#pragma omp parallel *[clause[[,]clause] ...]*

structured-block

The parallel directive specifies a parallel region in the code. The thread that encounters the *parallel* directive is the master thread. The role of the master thread is to create and start up a number of threads for executing the program in parallel within the parallel region. Every thread will have a unique id, an integer number. The master thread always has the id = 0, and the other threads spawned by the master get id = 1,...,*n*-1, where *n* is the number of threads in the team.

The optional clauses that can be used with the parallel directive are:

if(*scalar-expression*)	only create the region if the condition holds
num_threads(*integer-expression*)	specifies the max. number of threads to create
default(**shared** | **none**)	sets whether variables are shared by the threads by default
private(*list*)	specifies which variables are private to the threads
firstprivate(*list*)	specifies which private variable should be initialised
shared(*list*)	specifies which variables are shared among the threads
reduction(*operator*: *list*)	perform the specified reduction operation at the end of the region

The parallel construct is indispensable in a parallel OpenMP code but note this directive does not distribute the work among the threads. We will use other directives in combination with *parallel* directive. Without these *work sharing* directives, the code in the parallel region will be simply replicated, executed multiple times unnecessarily! This is illustrated by the next simple example.

```
#include <omp.h> // include for OpenMP
int main(int argc, char* argv[])
{
	printf("Starting the OpenMP sample program...\n");
	omp_set_num_threads(4); // use 4 threads
	printf("In master thread: starting parallel execution... \n");
	#pragma omp parallel
	{
		printf("in parallel region: in thread %d\n",
		omp_get_thread_num());
	}
	printf("In master thread: parallel execution finished... \n");
	return 0;
}
```

The following is the console output of the program. Note how the printf() function is executed multiple times, once by each thread in the team.

```
Starting the OpenMP sample program...
In master thread: starting parallel execution...
in parallel region: in thread 0
in parallel region: in thread 1
in parallel region: in thread 2
in parallel region: in thread 3
In master thread: parallel execution finished...
```

At the end of the parallel region there is an implied *barrier synchronization* that ensures that all threads finish their work before the master thread continues execution after the parallel region.

5.7.2 Sharing work among threads

A number of directives can be used to distribute work among the threads of the parallel region. These are the *for*, *sections*, *single*, *master* and *task* directives. In this section we will only focus on the *for* and the *sections* directives.

Loops

The *for* directive is used to distribute work within a **for** loop. The syntax of the directive is:

#pragma omp for *[clause[[,]clause] ...]*

for-loops

where the usable clauses are:

private(*list*)	as listed above in omp parallel
firstprivate(*list*)	as above
lastprivate(*list*)	as above
reduction(*operator*: *list*)	as above
schedule(*kind[*, *chunk_size]*)	specifies how work units are allocated to threads
ordered	the order of execution is the same as the loop order
nowait	skip the barrier synchronisation at the end and continue immediately

The result of the use of this directive is that iterations of the subsequent loop are executed by different threads of the team in parallel. If there are more iterations than threads, each thread will execute several iterations. The OpenMP implementations do not guarantee any order and allocation strategy in distributing the iterations to the thread except that each iteration will be executed only once. If we have specific requirements for the distribution strategy, we can use the additional **schedule** clause that we will discuss in the next section in detail.

Usage example:

```
printf("In master thread: starting parallel execution... \n");
#pragma omp parallel
{
      #pragma omp for
      {
            for (i=0; i<n; i++)
                  x[i] += y[i];
      }
}
printf("In master thread: parallel execution finished... \n");
```

If only one for loop is defined in the parallel region, we can use the *parallel for* compact form for the combined directives, as shown in the next code extract.

```
printf("In master thread: starting parallel execution... \n");
#pragma omp parallel for      //shorthand form
```

```
{
      for (i=0; i<n; i++)
            x[i] += y[i];
}
printf("In master thread: parallel execution finished... \n");
```

As we discussed earlier, by default, all threads can access all variables of the program; i.e. the variables are treated as shared variables. It is a good practice, however, not to assume any default behaviour, but explicitly specify which variable should be accessed in which way by the threads. Besides reducing the potential error situations, this can have significant effect on the achievable parallel performance as well.

To do so, first we use the **default(none)** clause listed for the **parallel** directive along with the **private** and **shared** clauses. The default clause instructs the system not to assume any default variable access strategy. The variables we intend to make available for each thread should be listed in the **shared** clause. Variables specified in the **private** clause will be private to the thread executing the iteration. A copy of the specified variable is created at the start of the thread and these copies may have different values in the different threads. We revisit our first example to illustrate this concept more clearly.

```
#pragma omp parallel if (n > threshold) \ shared(n,x,y) private(i)
{
      #pragma omp for
      for (i=0; i<n; i++)
            x[i] += y[i];
} /*-- End of parallel region --*/
```

In this example, we specify the problem size *n*, the *x* and *y* vectors as shared variables but keep the loop index variable *i* private. We also show here how the **if** clause is used to specify conditional parallel execution. The parallel region is only created if the condition of the **if** clause is true. In this example, if the size of the vectors is smaller than a predefined threshold, the program will run in sequential mode.

It is important to note that private variables within a parallel region are not initialised on entering the parallel region. Similarly, the value of private variables is undefined when exiting the parallel region. The **firstprivate**(*list*) clause should be used if we need to initialise a private variable with a value. The firstprivate will use the variable before the parallel region with the same name as the private variable listed in the parameter list of the clause and initialise the private variable with its value. The **lastprivate**(*list*) clause is used in a similar way, except this clause will guarantee that the last value of the listed private variables will be available for use after the parallel region.

Sections

We use the **for** directive for distributing identical tasks among a team of threads. There are situations when threads should execute different types of computations in parallel. The **sections** directive is designed for this purpose.

```
#pragma omp sections [clause[[,] clause] ...]
{
        [#pragma omp section
        structured-block]
        [#pragma omp section
        structured-block]
        ...
}
```

The following clauses can be used with the sections directive:

private(*list*)	as above
firstprivate(*list*)	as above
lastprivate(*list*)	as above
reduction(*operator*: *list*)	as above
nowait	as above

5.7.3 Scheduling threads

When discussing the **for** directive, we have ignored the question of how the execution is scheduled on the threads of the parallel region. If we do not specify the scheduling needs, the compiler and runtime system will choose one. Since proper execution scheduling can largely increase performance, it is best to explicitly specify the scheduling strategy to be used. The **schedule**(*kind[,chunk_size]*) clause is used for this where *kind* refers to one of the following supported strategies.

- **static:** In static scheduling, iterations of a loop are divided and grouped into blocks or chunks. The number of chunks depends the size of the chunk specified in the *chunk_size* parameter. For instance, a loop with 100 iterations and *chunk_size* of 10 will create 10 chunks. These chunks are assigned to threads in the team in round-robin fashion in the order of thread number, that is, thread 0 will execute iterations 0...9, thread 1 executes iterations 10...19, and so on.

- **dynamic:** The dynamic scheduling strategy treats the threads as a pool of workers. Each thread executes a chunk of iterations and when finished, will request another chunk until no chunks remain to be distributed.

- **guided:** The guided scheduling is similar to the dynamic one with the difference that the chunk size start large and is decreased to the specified *chunk_*size value towards the end of the execution. This scheduling works well for computations where the load can be difficult to balance properly. Starting with large chunks keeps threads busy and differences in execution times will be evened out by scheduling successively smaller and smaller tasks.

- **auto:** In the auto mode, the scheduling decision is delegated to the compiler and/or runtime system.

- **runtime:** The runtime strategy is a special case in which we leave the scheduling decision to be passed to the program at runtime. The advantage of this approach is the scheduling is not hard-coded into the application but can be changed easily through the OMP_SCHEDULE *type[,chunk]* environment variable.

5.7.4 Synchronization of threads

OpenMP provides implicit barriers at the end of every parallel region that automatically synchronises threads. In most cases, this is sufficient in our programs, but in cases when threads manipulate shared variables, we may need explicit synchronisation and protection to remove non-deterministic behaviour.

There are several synchronisation directives for explicit synchronisation. In case we need explicit barrier synchronisation, we can use the **#pragma omp barrier** directive. If we need to create a critical section to protect a set of shared variables as they are updated by multiple threads at the same time within a parallel region, we use the **critical** directive.

> **#pragma omp critical** *[(name)]*
>
> *structured-block*

This construct implements mutual exclusion and ensures that only one thread can execute its critical section. All other threads will wait for their turn. If the critical section only contains a single variable update, we can also use the atomic directive that executes variable updates as an atomic operation:

> **#pragma omp atomic**
>
> *expression-stmt*

5.7.5 Runtime library routines

Besides directives, OpenMP also provides a set of library routines in order to interact with the runtime system. For details, the OpenMP documentation should be consulted; here we look at the most important functions only.

int omp_get_num_threads(void);

Returns the number of threads in the current team.

int omp_get_thread_num(void);

Returns the ID of the encountering thread where ID ranges from zero to the size of the team minus 1.

int omp_get_num_procs(void);

Returns the number of processors available to the program.

The following two functions are important for measuring execution time of our program.

double omp_get_wtime(void);

Returns elapsed wall clock time in seconds.

double omp_get_wtick(void);

Returns the precision of the timer used by **omp_get_wtime**.

5.7.6 Parallel example: matrix-vector multiplication

We repeat our matrix-vector example at the end of this section and show the OpenMP implementation. The external for loop iterating over the rows of the matrix ***A*** and vector ***y*** is shared among the threads of the parallel region, the internal summation loop is performed by each thread.

```
#pragma omp parallel for default(none) shared(n,a,x,y) private(i,j)
for (i=0; i<n; i++)
{
       a[i] = 0.0;
       for (j=0; j<n; j++)
              y[i] += a[i][j]*x[j];
} /*-- End of omp parallel for --*/
```

Compare this implementation with the Java version in Section 5.6.8. and note the difference in size and readability between the two versions. The OpenMP version is free from the 'construction' code required in Java to manage the threads, and the original, sequential matrix-vector code can be used in the parallel version as OpenMP takes care of data partitioning. For these reasons, for creating shared-memory parallel numerical computations, OpenMP provides a cleaner, more manageable approach to parallelisation.

5.8 REFERENCES

Brinch Hansen, P. (1975). "The programming language Concurrent Pascal". IEEE Trans. Softw. Eng. 2 (June), 199–206.

[19] E. W. Dijkstra, "Cooperating sequential processes," in in *Programming Languages*, F. Genuys, Ed. Academic Press, 1968.

[20] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, no. 9, p. 569, Sep. 1965.

[21] D. E. Knuth, "Additional comments on a problem in concurrent programming control," *Commun. ACM*, vol. 9, no. 5, pp. 321–322, May 1966.

[22] G. L. Peterson, “Myths about the mutual exclusion problem,” *Inf. Process. Lett.*, vol. 12, no. 3, pp. 115–116, 1981.

[23] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[24] C. A. R. Hoare, “Monitors: an operating system structuring concept,” *Commun. ACM*, vol. 17, no. 10, pp. 549–557, Oct. 1974.

[25] P. B. Hansen, “Operating system principles,” Jan. 1973.

[26] A. Grama and V. Kumar, *Introduction to parallel computing*. New York, London :: Addison-Wesley,, 2003, p. 636.

[27] C.-P. Wen and K. Yelick, “A Survey of Portable Message Passing Libraries.” 1992.

[28] W. Gropp, *MPI : the complete reference. Vol. 2, The MPI-2 extensions*. Cambridge, Mass. ;: M.I.T.,, 1998, p. 350.

[29] *Occam 2 : reference manual*. Hemel Hempstead :: Prentice-Hall International,, 1988, p. 133.

[30] J. Geraint. and G. J. M. GOLDSMITH., *Programming in Occam 2.* New York; London :: Prentice Hall,, 1988.

[31] L. E. Cannon, “A cellular computer to implement the kalman filter algorithm,” Jan. 1969.

[32] G. . Fox, S. . Otto, and A. J. . Hey, “Matrix algorithms on a hypercube I: Matrix multiplication,” *Parallel Comput.*, vol. 4, no. 1, pp. 17–31, Feb. 1987.

[33] M. Cytowski, “Matrix Multiplication on Blue Gene/P.”

[34] S. F. McGinn and R. E. Shaw, “Parallel Gaussian elimination using OpenMP and MPI,” in *Proceedings 16th Annual International Symposium on High Performance Computing Systems and Applications*, 2002, pp. 169–173.

[35] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and distributed computation : numerical methods*. London :: Prentice-Hall International (UK),, 1989, p. 715.

[36] B. Wilkinson and C. M. Allen, *Parallel programming : techniques and applications using networked workstations and parallel computers*. Upper Saddle River, N.J. :: Prentice Hall,, 1999, p. 431.

6 MESSAGE PASSING PROGRAMMING

Distributed memory multiprocessor architectures do not allow the use of programming languages that rely on the shared memory concept. In a distributed memory system, processes executed by the processing elements must exchange control information and data by explicit message passing, similarly e.g. to the way we send emails to our friends. Message passing parallel programs are collections of *cooperating sequential processes*.

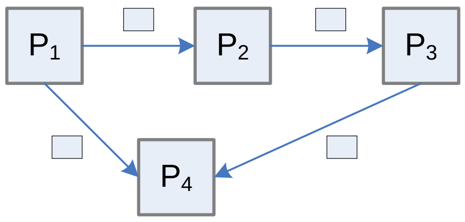


Figure 6-1 Processes cooperating by sending messages in a distributed memory message passing system.

While at first sight this programming method seems more complicated or unfamiliar than the shared memory approach, programs using message passing are typically more portable and prone to fewer errors since message passing eliminates all the problems caused by concurrent access to shared variables, and consequently the need for locking and synchronisation.

For instance, if we look at the producer-consumer problem using the bounded buffer from Section 5.4.2, and compare it to a message passing implementation, we can see that the communication channel is equivalent to the buffer. In the shared variable version we needed a shared buffer with necessary control mechanisms to provide correct communication between the processes. The consequence of using distributed memory is that the producer and consumer processes cannot access the same memory location. They communicate via the channel, which in turn – since it is not a shared variable – does not require mutual exclusion. Moreover, since data cannot be received before it is sent, condition synchronisation comes for free because of the FIFO property of the channel.

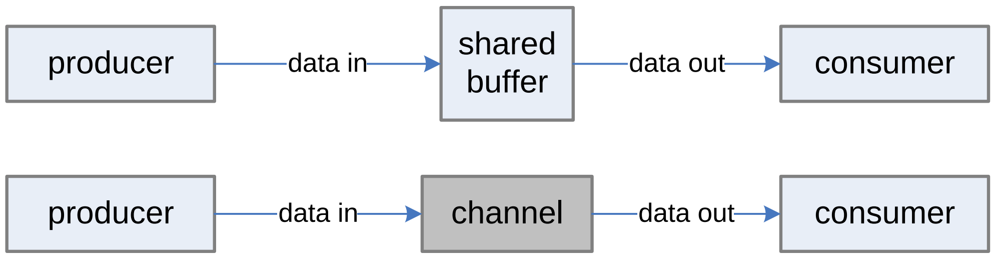


Figure 6-2 Comparison of the shared and distributed memory implementations of the Producer-Consumer problem.

With message passing, correct data transfer order and condition synchronisation is automatically provided. The following pseudo code of the message passing version of the producer consumer problem clearly shows how much simpler and understandable this version is when compared to any shared-memory implementation.

```
process Producer

loop begin
    produce data
    send data to consumer
loop end
```

```
process Consumer

loop begin
    receive value from producer
    consume data
loop end
```

Another advantage of message passing programs is that they are always executable on shared memory systems as well, since any communication channel can be mapped onto memory variables. This is not true the other way around; shared memory programs cannot run on distributed memory computers.

From the theoretical point of view, any message passing programming system must provide:

- operations for *sending* and *receiving* messages
- a mechanism for creating messages that can be transferred to other processes, and
- a way of identifying the source (receive) and destination (send) of the messages.

Since processes cannot access the memory of other processes, the message can only be sent to the other parties by using a data communication network (interconnect) infrastructure. Consequently, data is normally stored in memory during execution and sent over the channel only when required.

6.1 MESSAGE PASSING MODELS

Since the most important feature of message passing systems is the message-based communication, we start our overview of various message passing models. There are two fundamentally different ways in which messages can be exchanged between processes; the *synchronous* and *asynchronous* communication. The distinction is based on whether or not both parties must be ready for communication at the same time.

6.1.1 Synchronous message passing

In synchronous message passing we expect both parties to be ready for communication in order for the message exchange to take place. As shown in Figure 6-3, the sender process initiates the message transfer executing a send instruction. The instruction will block until the receiving process reaches and executes the corresponding receive instruction. The sender receives an acknowledgement for the connection establishment and the data is transferred. This type of communication is similar to a phone conversations. We cannot talk to the person we are calling until he or she answers the phone. Once the receiving process is ready, the connection is established and data can be transferred.

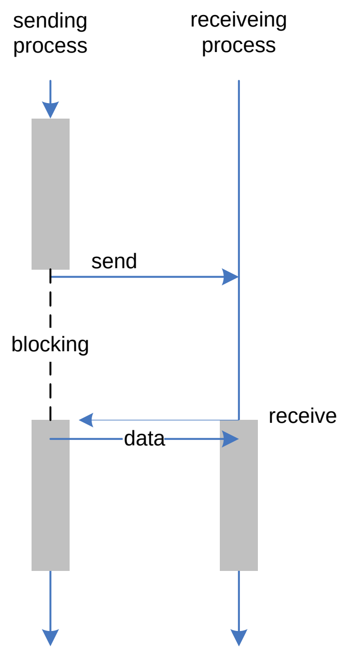


Figure 6-3 Space-time diagram of a synchronous message transfer operation.

The main *advantage* of the synchronous model is its simplicity. Since both parties are ready for communication, the channel can be implemented without intermediate buffering. This reduces the memory requirements of the implementation as well as reduces communication time by eliminating data copying from the buffers. Since both the sender and the receiver must be ready for communication to take place, this mode of communication *automatically synchronizes* the communicating processes to each other. The *disadvantage* of this communication method is the need to wait for the receiver. If the receiver is reaching the receive instruction long after the send instruction of the sender, the sender process will be blocked; the sender thus will not be able to do any useful work while waiting. Even worse, if the receiver process – for any reason – will never execute the receive instruction, the sender will *block indefinitely*; a deadlock situation will occur!

6.1.2 Asynchronous message passing

Asynchronous message passing differs from the synchronous one in that in this version, the communicating parties do not wait for each other in order to communicate. The sender can immediately send the message when it reaches the send instruction without checking the availability of the receiver. This is similar in nature to email communication. We send emails when we wish and accept that the receiver may receive and process the message some time later. This communication is only possible if the channel can store the message until it is delivered. The advantage of this approach is that the sender is never delayed on send. Having sent the message, the sender can continue doing useful work. Only the receiver is blocked until the message becomes available.

Asynchronous message passing has the following three properties:

- A channel acts as a FIFO queue for messages with infinite capacity.
- The sender is not delayed by a send.
- The receiver is delayed only if the channel is empty.

The two possible timing situations in message transfer are called *early send* and *early receive*. As shown in Figure 6-4, early send is the situation when the sender process reaches the send instruction before the receiver process executes the receive instruction. In this situation, the data is sent to the channel and becomes available when the receiver executes the receive instruction. Early send results in the best performance as none of the communicating parties need to wait. In the early receive situation, the receiver process is faster and executes the receive instruction before the send is reached. Consequently, the channel will be empty and the receiver will have to wait until the data becomes available. This means that early receive situations synchronise the two processes (processes start executing the code after the send and receive instructions approximately the same time) in contrast to the early send one in which there is no synchronisation.

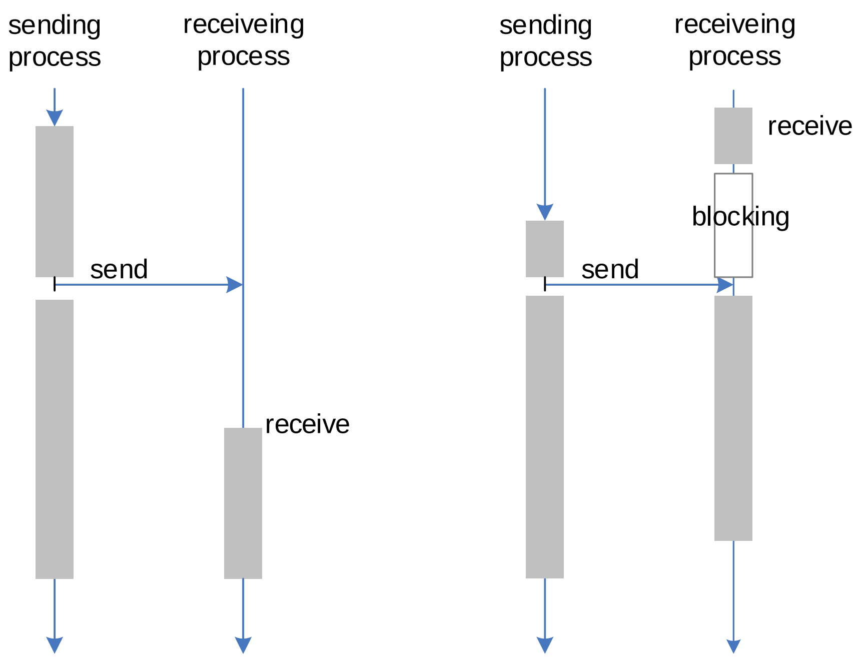


Figure 6-4 Blocking behaviour of the receiver depending on communication timing. "Early send" on the left, "early receive" on the right. The receiver is blocking only in the case of "early receive".

In practical systems, a channel with infinite buffer capacity cannot be implemented. To preserve memory, asynchronous message passing systems are implemented as buffered messaging systems, which use a finite size buffer for the intermediate storage of the messages awaiting delivery. The operation of these systems is a combination of asynchronous and synchronous message passing based on the following rules:

- The message is sent and the sender is not delayed if the channel buffer can hold the message.
- If the message is too large for the buffer, the sender waits (blocks) until the receiver becomes ready and the message is sent directly to the receiver without using the buffer.
- The receiver is delayed only if the channel is empty.

A major disadvantage and risk factor of asynchronous communication is when the receiver fails for some reason. Since the sender typically is not blocked waiting for the receiver to receive the message but continues execution, feedback about problems with message delivery or erroneous processing may reach the sender long after execution of the send instruction. This requires special error handling and state rollback mechanisms that make the program more complicated than in the synchronous case. Programmers typically use callback functions to handle responses and error situations.

6.1.3 Collective communication operations

There are situations when one process needs to communicate with more than one process. For example, it may need to send a number to all other processes in the program. Communication operations that involve more than one sender and receiver processes are called *collective communication operations* [26].

The following types of operations are used in parallel systems:

Broadcast A type of one-to-all communication in which one process sends the same data to all other processes of the program. An extension of the one-to-all broadcast is the all-to-all (or multinode) broadcast in which all processes perform a broadcast at the same time.

Scatter Scatter data from one member to all members of a group. This operation is used to send different parts of a larger data structure to different processes.

Gather The opposite of the scatter operation. Gathers data from all proceses to one process. A variation of gather is "allgather" where all members of the group receive the result.

Scatter/Gather Performs an all-to-all gather and scatter for all processes. It is also called as *complete exchange* or simply *alltoall* operation.

Reduction Collects results from all processes to one process and executes a specified operation at the same time. Common operations include *sum*, *max*, *min*, *average*.

Barrier synchronisation Barrier synchronization acts as traffic light for processes. It is used to make sure that processes continue their work only after each process has finished their previous step.

Each collective communication operation can be implemented using only the basic send and receive operations. In practice, this can lead to decreased performance as we show in the followings. Imagine that you need to send the same value for every process in a program, that is, perform a broadcast. Using only basic send/receive instructions, this could be implemented with a loop as follows:

```
loop begin
    send value to process i, i = 0,…,n-1
loop end
```

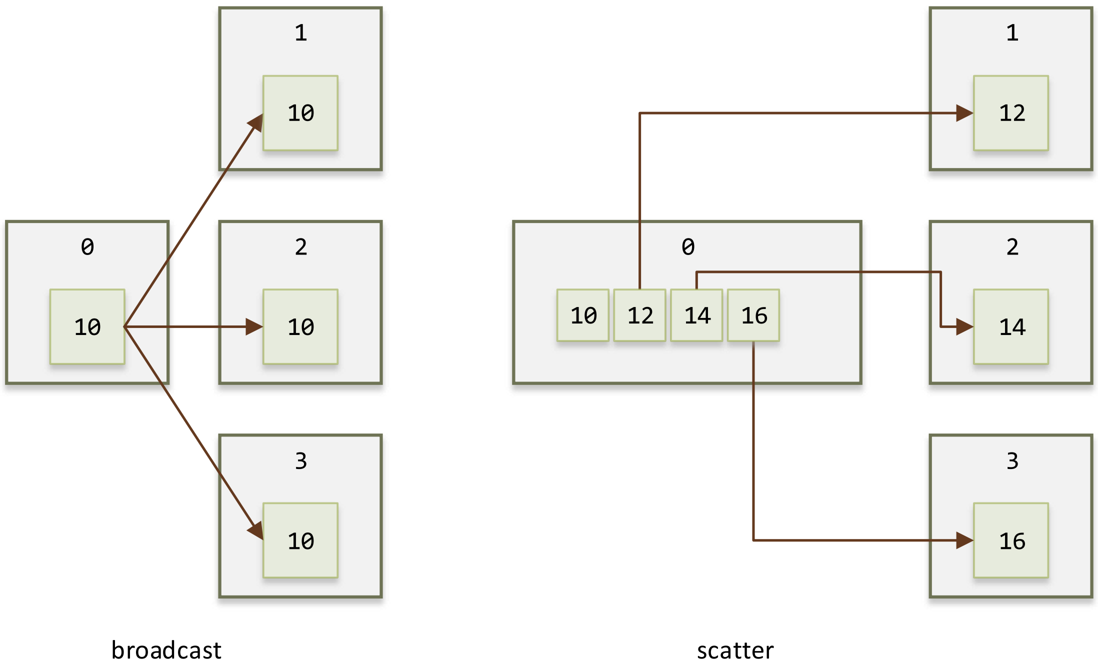


Figure 6-5 *Graphical illustration of the broadcast and scatter operations*

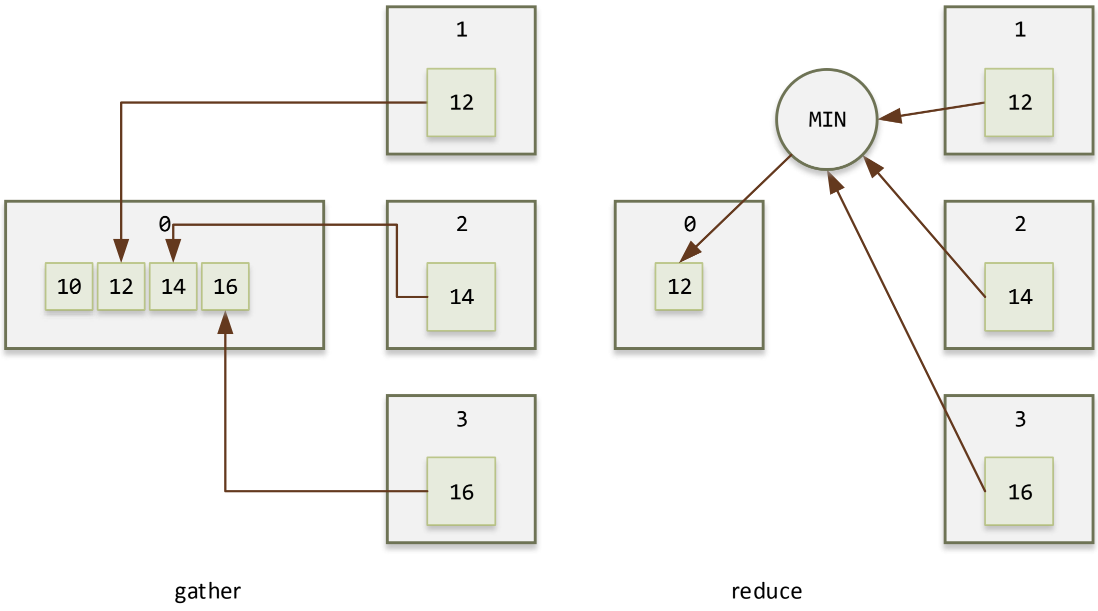


Figure 6-6 *Graphical illustration of the gather and reduction operations*

This implantation is an $O(n)$ operation; the more processes we have in the program, the more time it will take. This is the naïve implementation of the broadcast operation. Most modern parallel computers provide hardware support for broadcast and other collective communication operation. If we could have specific operations for these special communication operations, we could improve the performance and the readability of the programs as well.

6.1.4 Language support for message passing

Any programming language designed for the message passing style of parallel programming need constructs for managing and addressing processes, creating and sending/receiving messages in various ways, including support for collective communication operations. How this is provided and to what extent differentiates message passing programming languages.

Process management

The executing parallel program is a collection of processes. The creation of processes might be an implicit or explicit language keyword, a call to a library, or left to the runtime system. Static execution systems start processes at the start of the program and they cannot be changed or new processes cannot be created during execution. Dynamic systems allow for creating and destroying processes at run-time.

Process naming strategy

Another differentiating factor is how processes are identified in the system. Some use named processes, some use system level process IDs or unique logical IDs. Other strategies focus on the communication channel and either name the channel on which the message is forwarded to another process, or name the endpoint of the channel that is the local representation of the channel to the other process.

Communication strategies

The level of communication support also varies in message-passing systems. One differentiating factor is whether the language provides built-in keywords for messaging or uses external libraries. Another one is whether it supports only synchronous or asynchronous send/receive operations or both. It is also important when choosing a message-passing language or environment to have support for collective communication operations.

There are several approaches to providing support for parallel programming in programming languages. In some languages, parallelism is central to the language and is supported at the language level. The consequence of this approach is the development of new languages which is a positive result but at the same time it places extra burden on the parallel industry and software developers trying to keep up with the changes. A similar approach is to extend existing languages with new constructs, which has the advantages of relying on well-now syntax and tools. A third option is to use external libraries for parallel programming features. In the rest of this section, we will overview two examples of these approaches, the occam language (representing the first approach) and the Message Passing Interface (MPI) standard (representing the third approach). Due to its widespread acceptance and significance, we start our discussion with MPI.

6.2 MPI – The Message Passing Interface

By the end of the 1980s a number of message passing libraries had been developed and used [27]. Since several new models of parallel computers appeared in the market, serious portability problems occurred. Each manufacturer had its own message passing environment and users had to adapt their programs from environment to environment and from machine to machine. To put an end to this fragmentation, a group of researchers working on various message passing libraries started the development of a standard message passing library in 1992 which became published as the Message Passing Interface 1.0[17] in 1994.

MPI was meant to remain a specification that served as a guideline for vendors developing their own implementation of this standard. The goal was to ensure that users find the same environment on every message passing parallel computer with the same standardized functionality regardless of its architecture or manufacturer.

The first version of MPI defined the specification of traditional message passing. As experience with the system and its applications accumulated, more and more functionality was required by users. This led to the second version of MPI, MPI 2 [28] that introduced parallel I/O, dynamic process management and remote memory operations.

The designers of MPI took the approach of adding the message passing functionality as a library extending traditional programming languages. The standard provides bindings for Fortran and C languages. Over the years, other language bindings appeared, such as C++ and Java. In this book, for space consideration, we only discuss the C API. Since MPI is only a specification, it has many implementations, some open source some proprietary. One of the most widely used free implementation is the MPICH[18] library that allows, among other features, the use of individual computers typically found in university labs to be configured as one large multiprocessor environment. Details of the installation and operations can be found at the project home page.

[17] http://www.mpi-forum.org/docs/docs.html

[18] http://www.mpich.org/

6.2.1 Basic MPI

MPI functionality can be grouped into basic, intermediate and advanced level functionalities. The basic level is extremely simple; as we will see, one can create a parallel program with only a few additional function calls.

A minimal MPI program

The following simple example illustrates the simplest possible MPI program using only four MPI functions. The program will create a number of processes and print out their process IDs. To create an MPI program in C, we must import the `mpi.h` header file and link the compiled object code to the specific *mpi library* provided by the implementation. In our examples, we assume that a runtime system starts the processes. The number of parallel processes is irrelevant for the examples but can be specified when starting the programs.

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

int main(int argc, char *argv[])
{
    int numprocs, myid;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if (myid == 0)
    {
        printf("master process is running");
    }
    else
    {
        printf("slave process is running");
    }
    MPI_Finalize();
    return 0;
}
```

Let us discuss the used functions in turn. Every MPI program must include calls to so-called environment management functions. These functions ensure that the MPI runtime environment will know about the given process and that, in turn, will be able to communicate with the other processes. The MPI environment is a runtime infrastructure (also known as middleware) whose main responsibility is to provide the means for communication for the processes of our program. The set of functions we need for a minimal MPI program are the followings.

```
        int MPI_Init(int *argc, char ***argv);
```

MPI_Init registers the program in the MPI environment. This should only be called once at the beginning of each process.

```
        int MPI_Comm_size (MPI_Comm comm, int *size);
```

MPI_Comm_size returns the size of the process group associated with a communicator. In the default case, this is the number of processes in our program. MPI processes belong to groups. The communicator represents a process group and represents context as well as membership functionalities for processes. The default group including all processes of our program is represented be the communicator MPI_COMM_WORLD.

```
        int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

MPI_ Comm_rank returns the rank (ID) of the calling process in the MPI process group.

```
        int MPI_Finalize();
```

MPI_Finalize notifies the MPI environment that the process has finished and can be removed from the MPI process list. This should only be called once at the end of each process and must be called by every process.

Basic message passing

Beyond the four environment functions shown above, two additional functions are needed to create a fully functional message-passing parallel MPI program; these are the basic *send* and *receive* functions. MPI provides many variants of these operations but the basic versions are sufficient for getting started.

Send message

```
int    MPI_Send(void    *buf,    int    count,    MPI_Datatype    datatype,    int    dest,
              int tag, MPI_Comm comm)
```

The meaning of the parameters are:

buf	the memory address of the send buffer
count	the number of data elements in the send buffer (nonnegative integer) to be transferred
datatype	MPI datatype of the send buffer elements
dest	rank of the destination process
tag	message tag (integer), can be used to label messages
comm	communicator specifying process groups (default is all processes of the program: MPI_COMM_WORLD)

The message to be sent to another process must be prepared in memory before the function is called. This memory area acts as the send buffer. It has to be large enough to store the number of MPI data elements to be sent. Since MPI must work on heterogeneous systems as well, the messaging layer is responsible for delivering data in correct representations. To provide a neutral data representation in the program, special MPI data types must be used when packing data into message buffers. The mapping from C to the available MPI data types is shown in the following table.

C data type	MPI data type
char	MPI_CHAR
short int	MPI_SHORT
int	MPI_INT
long int	MPI_LONG
float	MPI_FLOAT
double	MPI_DOUBLE
long double	MPI_LONG_DOUBLE

Receive message

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
```

The meaning of the parameters are:

buf	memory address of the receive buffer into which the received data is stored
count	the maximum number of elements the receive buffer can store (integer)
datatype	datatype of each receive buffer element
source	rank of the source process
tag	message tag
comm	communicator
status	stores and returns status information about the execution of the receive operation

In the following simple example, we create a program consisting of two processes that exchange messages with each other in turn. This sample will illustrate the use of the basic send/receive MPI functions. As a practical comment, we add that this example also shows how to place the code for different parallel processes into one source file – for maintenance and clarity reasons, we prefer to place process code in a single source code file whenever possible – and use the rank parameter to differentiate behaviour for different processes (e.g. master vs. slave processes) as necessary.

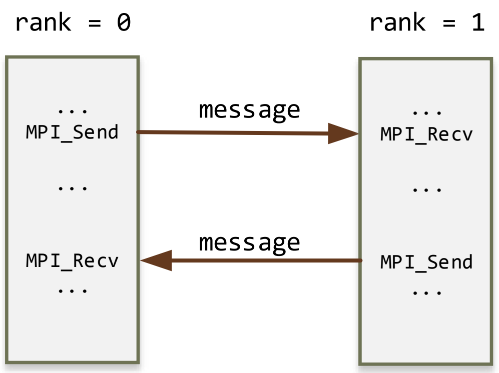


Figure 6-7 Structure of the Ping-Pong example

```
// A ping-pong MPI example program

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

int main(argc, argv)
int argc;
char **argv;
{
    int rank, size;
    int message[1];
    int i, j;
    MPI_Status status;
    int buf;
    int myrow, mycol;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    message[0] = 100;
    printf( "Ring sample program\n" );
    if (rank == 0) /* main process */
    {
        /* send first message */
        MPI_Send(message, 1, MPI_INT, rank + 1, 2, MPI_COMM_WORLD );
        MPI_Recv(&buf, 1, MPI_INT, size-1, 2, MPI_COMM_WORLD, &status);
        // print result
    }
    else
    {
        MPI_Recv(&buf, 1, MPI_INT, rank - 1, 2, MPI_COMM_WORLD, &status);
        MPI_Send(&buf, 1, MPI_INT, (rank + 1) % size, 2, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

These send and receive operations are blocking. The MPI_Recv will not return until the message has been received. The MPI_Send will also block but to understand correctly we must introduce a further concept, *completion*. MPI uses the concepts *locally complete* and *globally complete* to refer whether an operation has completed on the initiating side (locally complete) or all parties involved have completed their steps in the operation (globally complete). The MPI_Send blocks until the operation is locally complete. This means that after the send, the data buffer can be reused without affecting the message. Returning from the send, however will not guarantee that the receiving end has actually received the message.

MPI uses so-called communication modes to alter the behaviour of the send/receive operations. These modes are:

Standard	The standard mode, send will block only until the buffer can be reused.
Synchronous	Requires the send to wait for the completion of the matching receive operation. The two processes complete together, hence synchronise their execution sequence.
Buffered	This mode allows the user to explicitly control message buffering.
Ready	In this mode, the send may be started only if matching receive has already been reached. This allows the system to perform communication the fastest way but it is very easy to make mistakes using this mode.

These modes can be specified by using additional MPI functions whose name is created by adding a letter to the Send/Receive function names, e.g.:

MPI_Ssend	Synchronous blocking send
MPI_Bsend	Buffered blocking send
MPI_Rsend	Blocking ready send

Ring example

We continue our discussion with a further communication example. In this program, the processes form a virtual ring topology and only communicate with their immediate neighbours. The structure of the program is shown in the next figure. Process with rank = 0 is the master that starts the communication sequence. It sends a message to the next process with rank = 1. Process 1 will receive the message and pass it on to process 2. The sequence continues until the last process (rank = n-1) receives the message and sends it back to the master. Note the use of the *modulo* arithmetic in specifying the destination addresses.

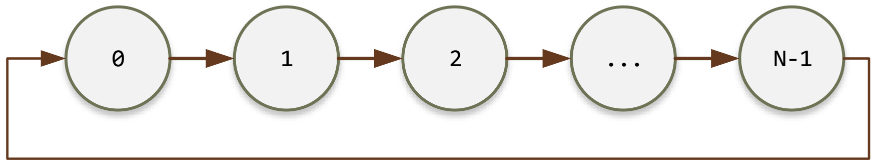


Figure 6-8 *Communication paths in the ring of processes example program.*

```
// A ring topology MPI example program

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


int main(argc, argv)
int argc;
char **argv;
{
    int rank, size;
    int message[1];
    int i, j;
    MPI_Status status;
    int buf;
    int myrow, mycol;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    message[0] = 100;
    printf( "Ring sample program\n" );
    if (rank == 0) /* main process */
    {
        /* send first message */
        MPI_Send(message, 1, MPI_INT, rank + 1, 2, MPI_COMM_WORLD );
        MPI_Recv(&buf,    1,    MPI_INT,    size-1,    2,    MPI_COMM_WORLD,
                  &status);
        // print result
    }
    else
    {
        MPI_Recv(&buf,   1,   MPI_INT,   rank   -   1,   2,   MPI_COMM_WORLD,
                  &status);
        MPI_Send(&buf,   1,   MPI_INT,   (rank   +   1)   %   size,   2,
                  MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

MPI has a lot more to offer than the six functions we have used so far in our examples. At a more advanced level, interesting new features are available for creating more efficient and maintainable parallel programs. In this part we will look at three areas of functions, namely creating virtual topologies, performing collective communication and using non-blocking send/receive operations.

6.2.2 Virtual topologies

When we partition a data set for data-parallel computation, we create an application level topology. This topology defines the topology of the different data segments. Typically, we can have one, two and three-dimensional application topologies. Figure 6-9 illustrates some of them. The data segments will be mapped to processes that consequently will form the same logical topology. The actual hardware, however, may have a different interconnection topology than the one we created based in the application, therefore we need an extra step, to map the logical topology (our processes) to the physical topology (the processors). This mapping can be difficult if we do not know the exact topology used in the parallel computer in question. This is frequently the case as only the manufacturers know the best embedding for a particular topology for their computer. E.g., it may seem logical to use the process with the next larger rank as a neighbour but the hardware interconnect may provide a faster path to some other process instead. The logical topology hence should not be hard-wired by rank numbers.

MPI provides four functions to map our topology to the machine in a platform neutral way: `MPI_Cart_create` to create a virtual topology, `MPI_Cart_shift` to find neighbour processes, `MPI_Cart_get` to retrieve the topology information and `MPI_Cart_coords` to get the coordinates of process with a particular rank. The word Cart refers to the Cartesian coordinate system these functions use in describing the topologies.

Let us now see the functions in detail.

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],
                    const int periods[], int reorder, MPI_Comm *comm_cart)
```

Creates a new Cartesian topology. The meaning of the parameters are:

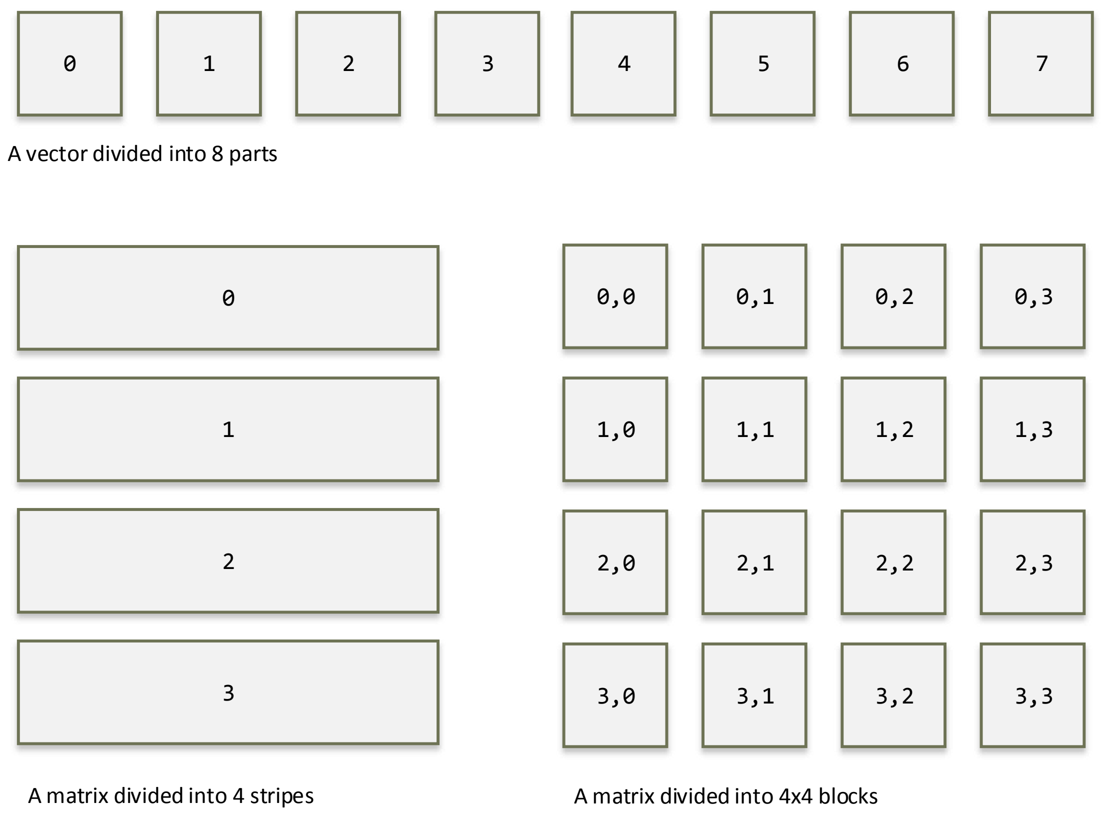

0
1
2
3
4
5
6
7
A vector divided into 8 parts
0
1
2
3
A matrix divided into 4 stripes
0,0
0,1
0,2
0,3
1,0
1,1
1,2
1,3
2,0
2,1
2,2
2,3
3,0
3,1
3,2
3,3
A matrix divided into 4x4 blocks

Figure 6-9 Various application topologies based on data partitioning

comm_old	input communicator
ndims	number of dimensions of Cartesian grid (typically 1, 2 or 3)
dims	integer array of size *ndims* specifying the number of processes in each dimension
periods	logical array of size *ndims* specifying whether the grid is periodic (true) or not (false) in each dimension
reorder	ranking may be reordered (true) or not (false)
comm_cart	the new communicator that will represent the new Cartesian topology

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
                    int *rank_source, int *rank_dest)
```

Returns the shifted source and destination ranks, given a shift direction and amount. The parameters are:

comm	communicator with Cartesian structure
direction	coordinate dimension of the shift
disp	displacement (> 0: upwards shift, < 0: downwards shift)
rank_source	rank of source process
rank_dest	rank of destination process

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],
                 int coords[])
```

Retrieves Cartesian topology information associated with a communicator. The parameters are:

comm	communicator with Cartesian structure
maxdims	length of vectors *dims*, *periods*, and *coords* in the calling program
dims	number of processes for each Cartesian dimension
periods	periodicity (true/false) for each Cartesian dimension
coords	coordinates of calling process in Cartesian structure

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
```

Determines process coordinates in Cartesian topology given rank in group. The parameters are:

comm	communicator with Cartesian structure
rank	rank of a process within the group of *comm*
maxdims	length of vector *coords* in the calling program
coords	integer array (of size *ndims*) containing the Cartesian coordinates of specified process

As an illustration of using the topology feature of MPI, we revisit the ring communication example shown above. Remember, we used the rank for addressing the neighbour process. This might not be the most efficient on some architectures. The following code sample shows how to get the neighbour address in a architecture neutral manner.

```
#include "mpi.h"

int main( argc, argv )
int argc;
char **argv;
{
    int rank, value, size, false=0;
    int right_nbr, left_nbr;  // rank of left and right neighbours
    MPI_Comm   ring_comm;     // communicator for the ring topology
    MPI_Status status;

    MPI_Init( &argc, &argv );

    MPI_Comm_size( MPI_COMM_WORLD, &size );
    // create a periodic topology with wrap-around connection
    MPI_Cart_create(     MPI_COMM_WORLD,     1,     &size,     &true,     1,
                  &ring_comm );
    // get my left and right neighbour ranks
    MPI_Cart_shift( ring_comm, 0, 1, &left_nbr, &right_nbr );
    MPI_Comm_rank( ring_comm, &rank );
    MPI_Comm_size( ring_comm, &size );
    do {
       if (rank == 0) {
          scanf( "%d", &value );
          MPI_Send( &value, 1, MPI_INT, right_nbr, 0, ring_comm );
          MPI_Recv( &value, 1, MPI_INT, left_nbr, 0, ring_comm,
              &status );

      }
      else {
          MPI_Recv( &value, 1, MPI_INT, left_nbr, 0, ring_comm,
              &status );
          MPI_Send( &value, 1, MPI_INT, right_nbr, 0, ring_comm );
      }
      printf( "Process %d got %d\n", rank, value );
    } while (value >= 0);
    MPI_Finalize( );
    return 0;
}
```

6.2.3 Collective communications

MPI provides a number of collective communication operation functions to simplify our programs and provide access to more efficient communication facilities in the underlying hardware. The functions we will discuss are the followings:

MPI_Barrier	synchronize all processes
MPI_Bcast	send the same data to all processes
MPI_Gather	gather data from all processes
MPI_Scatter	scatter data to all processes
MPI_Reduce	reduction operation

These operations are symmetric, which means that all processes execute exactly the same function. E.g. for the broadcast operation to execute, each process must call the MPI_Bcast function. The system will work out who will send and who will receive the data when executing the function.

The signatures of the functions are as follows. Parameters are only explained if not described with functions earlier.

```
int MPI_Barrier( MPI_Comm comm )
```

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,
                MPI_Comm comm )
```

The `root` parameter specifies the rank of the process broadcasting the data in the buffer.

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void    *recvbuf,    int    recvcount,    MPI_Datatype    recvtype,
                int root, MPI_Comm comm)
int  MPI_Reduce(const  void  *sendbuf,  void  *recvbuf,  int  count,  MPI_Datatype
                datatype, MPI_Op op, int root, MPI_Comm comm)
```

where op parameter is the reduction operator of type MPI_Op selected from the following pre-defined operators:

Name	**Meaning**
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

The following example demonstrates the use of the broadcast, scatter and reduction operations. The program will find the maximum difference between the individual values in a set of numbers stored in an array and predefined number. The first step of the program is to distribute the array among the processes using a scatter operation. Then the predefined number is broadcast to the processes that perform the local difference calculation. The local maximum difference values are gathered using a MAX operator in the reduction.

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

```
#include <malloc.h>

int main(int argc, char *argv[])
{
    int numprocs, myid;
      int data_size, root = 0;
      double *data;
      double value;
      const int length = 10000;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if (myid == 0)
    {
             data = (double*)malloc(length*sizeof(double));
             for (int i=0; i< length; i++){
                    data[i] = 100.0 * rand();
             }
             data_size = length / numprocs;
      }
      // broadcast data segment size
      MPI_Bcast( &data_size, 1, MPI_INT, root, MPI_COMM_WORLD);
      // broadcast value to get difference from
      MPI_Bcast( &value, 1, MPI_DOUBLE, root, MPI_COMM_WORLD);
      if (myid != 0){
             data = (double*)malloc(data_size*sizeof(double));
      }
      //scatter data
      int recv_size = data_size;
      MPI_Scatter(&data,  length,  MPI_DOUBLE,  &data,  recv_size,  MPI_DOUBLE,
                    root, MPI_COMM_WORLD);
      // compute
      double local_max = 0;
      double global_max = 0;
      for (int i=0; i< data_size; i++)
      {
             if (fabs(value - data[myid * data_size + i]) > local_max)
                    local_max = fabs(value - data[myid * data_size + i]);
      }
      // reduce
      MPI_Reduce(&local_max, &global_max, 1, MPI_DOUBLE, MPI_MAX, root,
                    MPI_COMM_WORLD);

      MPI_Finalize();
    return 0;
}
```

6.2.4 Non-blocking send/receive operations

For special cases MPI provides non-blocking send/receive routines. These operations will return immediately, regardless whether the send buffer can be safely reused or whether the receive statement has received the message. The non-blocking operations can be used to overlap communication with computation, which can greatly improve performance if communication delays are significant.

The routines for non-blocking communication are:

```
int  MPI_Isend(void  *buf,  int  count,  MPI_Datatype  datatype,  int  dest,
           int tag, MPI_Comm comm, MPI_Request *request)
```

```
int  MPI_Irecv(void  *buf,  int  count,  MPI_Datatype  datatype,  int  source,
           int tag, MPI_Comm comm, MPI_Request *request)
```

where all but the last parameter are the same as in the blocking case. The request …

The letter I in the routine name stands for *Immediate*, referring to the immediate return from the call. If the routines do not block, how can we detect that our message has been sent and delivered safely? Two additional routines can be used for this verification step.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

will wait for completion of the operation specified by the *request* variable returned from the non-blocking send/receive routine. If we do not want to block with the wait routine, we can quickly test the status of the send/receive operation by calling the function

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

which will return immediately and indicate with the `flag` variable whether or not the send/receive is complete.

The four different communication modes (standard, synchronous, buffered and ready) can also be used with the non-blocking send. The routine MPI_Issend() – note the two 's' characters – will return immediately from the send but when the message is delivered, the sending and receiving processes synchronise for the message transfer. For the receive routines, only the standard mode is allowed.

6.2.5 Matrix-vector multiplication

We close the overview of MPI with the matrix-vector multiplication example used earlier with Java and OpenMP. The purpose of this example is to compare the shared memory implementations with a message passing one. We will use most of the communication features of MPI for this example. We will use the same row-wise striped data distribution strategy as in the previous examples.

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

int main(int argc, char *argv[])
{
    double starttime, timerres;
    int size, rank, scatterSize;
    float **A, *x, *y;
    int dim = 100;  // default dimension
    int root = 0;   // root rank for broadcast

    if (argc == 2)
        dim = atoi(argv[1]);
    else
        dim = 100;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size );
    MPI_Comm_rank(MPI_COMM_WORLD, &rank );

    timerres = MPI_Wtick();
    starttime = MPI_Wtime();
    if (rank == 0)
    {
        // allocate memory for matrices
        A = (float*)malloc(dim * dim * sizeof(float));
        x = (float*)malloc(dim * sizeof(float));
        y = (float*)malloc(dim * sizeof(float));
        // initialise matrices
    }
    // broadcast dimension, process number
    MPI_Bcast(&dim, 1, MPI_INT, Root, MPI_COMM_WORLD);
    MPI_Bcast(&size, 1, MPI_INT, Root, MPI_COMM_WORLD);

    // allocate memory for vector x on the other processes
    if (rank != 0)
        x = (float *)malloc(VectorSize*sizeof(float));
    // broadcast vector x
    MPI_Bcast(x, dim, MPI_FLOAT, Root, MPI_COMM_WORLD);
    // scatter matrix A
    scatterSize = dim / size;
    Mybuffer = (float *)malloc(scatterSize * dim * sizeof(float));
    MPI_Scatter( Buffer, scatterSize * dim, MPI_FLOAT, Mybuffer,
        scatterSize * dim, MPI_FLOAT, 0, MPI_COMM_WORLD);
    // create vector y segments
    MyFinalVector = (float *)malloc(scatterSize*sizeof(float));

    for(irow = 0 ; irow < scatterSize ; irow++) {
       MyFinalVector[irow] = 0;
       index = irow * dim;
       for(icol = 0; icol < dim; icol++)
       MyFinalVector[irow] += (Mybuffer[index++] * x[icol]);
    }

    if( MyRank == 0)
        FinalVector = (float *)malloc(dim*sizeof(float));
    // gather vector y segments into final result vector on master
    MPI_Gather( MyFinalVector, scatterSize, MPI_FLOAT, FinalVector,
           scatterSize, MPI_FLOAT, Root, MPI_COMM_WORLD);

    stoptime = MPI_Wtime();
    //printf("exec time = %.3f", (stoptime-starttime)/timerres!!);
    MPI_Finalize();
}
```

6.3 OCCAM

In this section, which is a suggested reading for interested students, we take a short look at the occam programming language [29] primarily for educational and historical reasons. Our purpose is not to give a full coverage but only to introduce philosophy and the key features of the language. The occam language was developed in the early 1980s as an implementation of Prof Hoare's CSP theory (Communicating Sequential Processes) [23] and it is one of the few languages that have been designed from the ground up for writing parallel programs.

The language took its name from the English philosopher William of Ockham (also written as Occam; c. 1287 – 1347) known for Occam's Razor stating "*entities must not be multiplied beyond necessity*" [29]. The concept of simplicity is characteristic to this language.

The key concept of occam is the process, which is a sequential program segment with its own state, and the channel through which processes communicate with each other via explicit message passing. Processes can be combined to create more complex processes. Occam is a very safe language due to its strong theoretical (formal) foundation and to its strict syntax, strong typing and static variables. Pointers and dynamic data structures are unknown to the language, consequently memory addressing error can be detected during program compilation.

6.3.1 Introduction

The language is based upon five *primitive* processes. These instruction level processes are called *actions* and are as follows:

action	***occam example***	
Assignment, :=	a:=2	
output (send), !	ch ! 3	send the value 3 on channel 'ch'
input (receive), ?	ch ? x	assign value received from 'ch' to x

Note that the send ! and receive ? actions are language level constructs, not added-on library calls. Message passing is literally at the core of this language. There are two further building blocks, the primitive processes SKIP and STOP.

process	***meaning***
SKIP	*do nothing*

STOP	*stop and DO NOT continue*

The last, STOP process is not for terminating a program, such as e.g. the 'exit()' function in C. STOP breaks the execution of the program at the given point and forces it into an infinite wait – the program will never terminate! The main use of STOP is in the debugging stage to stop at errors and make them detectable for the debugger.

6.3.2 Language constructs

To create real programs, these five primitive processes can be combined to add structure to the program. Any occam program is a hierarchical structure of processes. Larger processes created from smaller ones are called *constructions*. The created compound processes are one of the following kind:

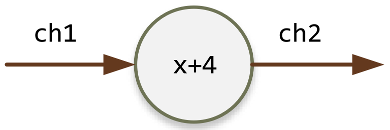
SEQ	sequential
IF	conditional
CASE	selection
WHILE	loop
PAR	parallel
ALT	alternative

The first four types are used to create sequential processes. The PAR and ALT are used to build parallel processes that communicate via channels. The SEQ, IF, PAR, ALT can be replicated, i.e. create multiple copies of the same process.

The syntax of an occam program is very strict. Instead of braces, code blocks are designated by tabulation in the form of space characters. Each block is indented to the right by two spaces. Although this indentation rule was heavily criticised in its time, it is used in several modern languages, e.g. in Haskell and Python.

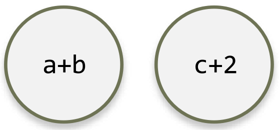
We will use a set of examples to illustrate the use of the above constructs. The first example process will receive an input value from channel `ch1`, increment the value by 4 and pass it on to another process via channel `ch2`. The order of the three statements (primitive processes) is specified by the SEQ construct. Since all statements are indented by two spaces identically, they are within the scope of the `SEQ` and executed in order.

```
SEQ
  ch1 ? x
  x := x + 4
  ch2 ! x
```



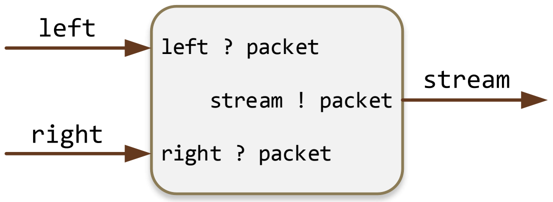
The next example will execute two assignments in parallel. The parallel execution of the two assignments is specified by the PAR construct. Note that there is no extra statements for creating and starting the parallel processes. The PAR construct handles all these details behind the scenes.

```
PAR
  x:= a+b
  y:= c+2
```



The third example shows the use of the special alternation construct of occam. Alternation selects from guarded processes depending whether they are ready. The guard is the input followed by the indented process to be executed if the guard executes. We illustrate the operation of the ALT construct with the following example. A process receives packets from two inputs channels (left and right) merges the packets as they are sent to the output stream channel. If the left channel is ready (there is data) the left packet is input and sent to the stream. If the right one is ready and the left is not, the right packet is input and forwarded. If none of the channels are ready, the process will block until a packet arrives on either channel. If both have data, one is selected at random.

```
ALT
  left ? packet
    stream ! packet
  right ? packet
    stream ! packet
```



Several other control constructs are available to create more complex programs. The most important ones are the conditional, loop, replicated constructs and the procedure. The conditional is most similar to the C switch statement but built from conditional expressions.

```
IF
  conditional expression
    process
```

```
      conditional expression
        process
```

Example:

```
IF
  x <> 0
    ch1 ! x
  TRUE
    SKIP
```

There are two basic loops in occam, the *while* and *for* loops. The syntax of the while loop is the following.

```
WHILE conditional expression
  process
```

The body of the process can be defined by the SEQ, PAR or ALT constructs as shown in the next example of a sequential loop. This example uses the SEQ within the loop.

```
x:= 50
WHILE x <> 0
  SEQ
    ch1 ! x
    x:= x-1
```

The for loop is an example of the *replicated constructs* of occam. The SEQ, PAR and ALT constructs can all be replicated. If the FOR keyword is used with the SEQ construct, a replicated SEQ is created that specifies how many processes are created that execute sequentially.

```
SEQ loop variable FOR repetition
  process
```

An example for the use of this construct is the a loop that adds the first 10 numbers:

```
SEQ i:=0 FOR 10
  x:= x+i
```

The replicated PAR also creates a specified number of processes but they will be executed in parallel.

```
PAR loop variable FOR repetition
  process
```

The following example will create five processes each calculating the square of a number in parallel.

```
PAR i:=0 FOR 5
  x[i] := i * i
```

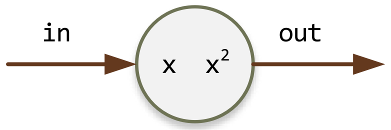
Procedures

To make large programs modular, readable and maintainable, occam allows the use of procedures. The syntax of the procedure declaration is as follows. Note the colon sign that marks the end of the procedure.

```
PROC name (parameters)
  process
:
```

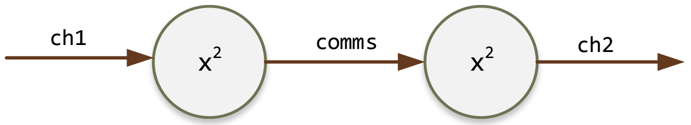
Example for a procedure declaration. This also shows a channel declaration in the formal parameter list. The CHAN OF INT specifies and input channel that can receive a single integer value at a time. This procedure creates a named process that can be a building block of further processes.

```
PROC square(CHAN OF INT in, out)
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      out ! x*x
:
```



Once we have a named process, we can use it to create more complex process structures. In this example we create a pipeline of two 'square' processes, producing the x^4 on the output channel ch2.

```
CHAN OF INT comms:
PAR
  square(ch1, comms)
  square(comms, ch2)
```



There are many other details and subtleties of the language but these are not vital in understanding the general concept. We complete our journey to occam with a program that creates a process framework for master-slave task execution. The program is a simplified version of the program described in [30]. The system uses four slave processes working in parallel. They are connected to the master via channels, declared as a channel array. The master keeps sending tasks to the slaves, receives the result from the fastest processes and sends it a new task in turn. The process repeats until there is no more work to, when a termination command is sent to the workers. The task, result and computation are indicated as italicised pseudo code, in a real implementation they should be properly declared.

```
-- channel declaration
[4] CHAN OF INT to.worker:
[4] CHAN OF INT from.worker:
-- main program
PAR
  farmer(to.worker, from.worker)
  PAR i = 0 FOR 4
    worker(i, to.worker[i], from.worker[i])

-- farmer process
PROC farmer([]CHAN OF INT to.worker, []CHAN OF INT from.worker)
  -- give one task to each process
  PAR i = 0 FOR 4
    to.worker[i] ! a task
   -- get the result and keep sending more tasks
   SEQ i = 4 FOR number of tasks - 4
     ALT j = 0 FOR 4
       from.slave[j] ? result
         to.slave[j] ! new task
   -- receive the last 4 results and send termination signal
   SEQ i = 0 FOR 4
     ALT j = 0 FOR 4
       from.slave[j] ? result
         to.slave[j] ! terminate

:
```

```
-- worker process
PROC worker(CHAN OF INT from.master, CHAN OF INT to.master)
  BOOL more:
  SEQ
    more := true;
    WHILE more
      from.farmer ? CASE
        new task:
          SEQ
            result := perform(new task)
             to.farmer ! result
        terminate:
          more := FALSE
    SEQ
      in ? x
      out ! x*x
:
```

[26] A. Grama and V. Kumar, *Introduction to parallel computing*. New York, London :: Addison-Wesley,, 2003, p. 636.

[27] C.-P. Wen and K. Yelick, “A Survey of Portable Message Passing Libraries.” 1992.

[28] W. Gropp, *MPI : the complete reference. Vol. 2, The MPI-2 extensions*. Cambridge, Mass. ;: M.I.T.,, 1998, p. 350.

[29] *Occam 2 : reference manual*. Hemel Hempstead :: Prentice-Hall International,, 1988, p. 133.

[30] J. Geraint. and G. J. M. GOLDSMITH., *Programming in Occam 2.* New York; London :: Prentice Hall,, 1988.

7 Parallel Algorithms

The purpose of using parallel computing is to solve problems that could not be solved on a sequential computer or solve them in much shorter time. The different parallel languages, computer architectures, tools are being developed to reach this goal. But they are only tools in the final battle, to describe the solution of a problem in a way that can be efficiently executed in parallel.

All the fundamental algorithms we know have sequential roots. In the paper and pen world, scientists had worked in a pure sequential "mode" dictated by the nature of human thinking. In the computer world, the first 40 years were dominated by sequential computers. Turning sequential algorithms into parallel one is not always simple. We can find a parallel implementation to almost any problem, the main question is whether it will be executing efficiently and achieve any useful speedup. This depends on an intricate combination of many factors: architecture properties, the chosen language, the approach to parallelisation strategy and the quality of implementation.

Most experts of parallel computing agree that successful parallel computing requires a different mind-set, a way of thinking that is different from the one we use in the design of sequential algorithms. To develop this way of thinking requires time and practice in the world of parallel algorithms. In this chapter, we discuss important algorithms and their parallel versions in a hope that this brings students closer to this thinking "in parallel" and to parallel programing in general.

7.1 Dense matrix algorithms

Traditionally, the key use of parallel computers is to solve large numerical problems. Many of these are based on the manipulation of dense matrices. The favoured approach to parallelism is almost always data-parallelism as it provides good scaling for a large number of processors (since the data sets are always much larger than the number of practically available processors). What we are interested in is to find where the parallelism is in the solution. While the particular implementation depends on the choice of architecture and language, in this chapter we plan to describe the parallel algorithms at a more abstract, language and architecture neutral manner.

Dense matrix problems include matrix manipulation (e.g. transposition), matrix operations (addition, multiplication, etc.) and solving systems of linear equations or partial differential equations. In this section we look at the first two and discuss equations separately. Before starting our discussion of the algorithms, however, we revisit data partitioning which is central to this class of algorithms.

7.1.1 Data partitioning schemes

We have already looked at possible data partitioning approaches earlier in this book. Here we mainly summarise these methods. The two main options we can use for matrices are the *striped* and *checkerboard* partitioning methods.

The striped approach can be further divided into row-wise (horizontal) or column-wise (vertical) stripes. In the row-wise case, the matrix is divided into groups of complete rows, in the column-wise case into groups of complete columns. These groups are then assigned or mapped to processors. Depending on how this is performed we distinguish between block and cyclic striping. Figure 7-1 illustrates the difference between the row-wise block and striping methods. In the block striping approach the processors receive a contiguous block of rows. In the cyclic approach, the rows of the matrix are assigned to the processors in a sequence in a wraparound manner. In the example, the first four rows are assigned to the four processors, then we start mapping the next four rows and continue in this manner.

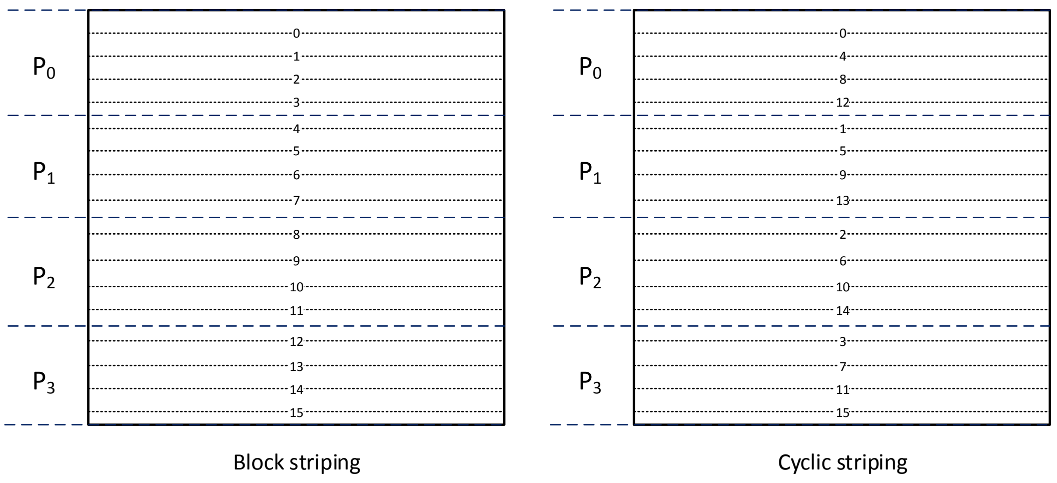
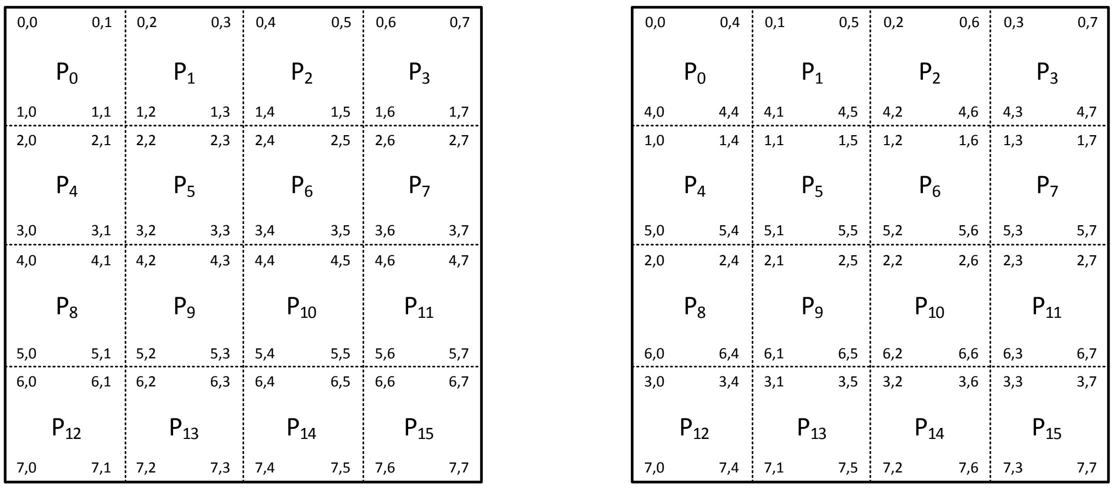


Figure 7-1 Illustration of the row-wise block and cyclic striping partitioning methods

In the checkerboard partitioning, we divide the matrix into smaller square or rectangular blocks. None of the processors receive a complete row or column. The submatrices can be mapped onto the processors, either in a block or cyclic fashion. The block checkerboard partitioning scheme is often called simply "block partitioning".



Block checkerboard partitioning

Cyclic checkerboard partitioning

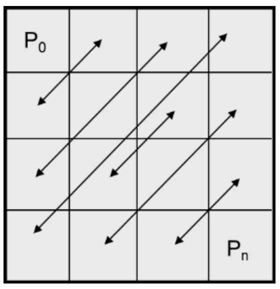
Figure 7-2 Illustration of the row-wise block and cyclic checkerboard partitioning methods

Several of these partitioning methods can appear within one parallel program since certain operations are better suited to one partitioning than to others. It can also happen that the same matrix must be repartitioned several times as the operations are executed in the algorithm.

7.1.2 Matrix transposition

Matrix transposition is a very simple but frequently occurring operation. To create the transpose $\mathbf{A}^T$ of matrix **A**, the elements of the matrix need to be reflected over the main diagonal of the matrix, i.e. one must exchange element a_{ij} with a_{ji}. The sequential algorithm for computing the transpose is as follows.

```
for (i=0; i<n-1; i++)
{
    for (j=i+1; j<n; j++)
    {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}
```



The algorithm works in row major order. The outer loop iterates over the rows of the matrix, while the inner loop iterates over the elements of each row found right to the main diagonal. The core of the loop performs the element swap operation.

In a shared memory environment it would seem natural to follow this algorithm. We can make the outer loop parallel (row-wise striping) and let each process execute the inner loop. The disadvantage of this method is that the work is not distributed evenly among the processors. Processors with higher index will get less and less work due to the triangular pattern of the inner loop. In OpenMP, this can be compensated by using dynamic scheduling but this will be only efficient if the dimension of the matrix is much larger than the number of processors.

```
#pragma omp parallel for schedule(dynamic, 1)
for (i=0; i<n-1; i++)
{
    for (j=i+1; j<n; j++)
    {
        // swap operation
    }
}
```

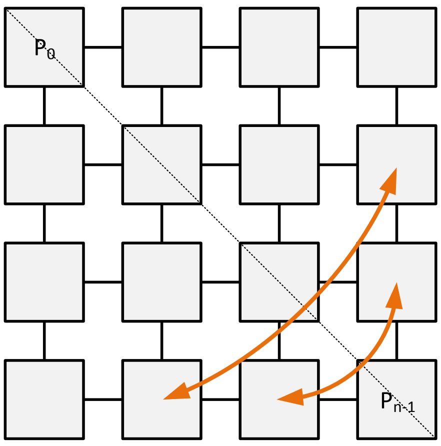
Another approach is the restructure the algorithm and create a sequence of swap operations for the upper triangular part of the matrix. A single parallel for loop can iterate over the list of swap operations. To make this work, we need an additional index array that helps picking up the coordinates of the elements that should be swapped.

```
// init the index array
int index[elements][2];
int ix = 0;
for (int r=0; r<n-1; r++)
{
        for (int c=r+1; c<n; c++)
        {
                index[ix][0] = r; // row
                index[ix][1] = c; // col
                ix++;
        }
}
```

```
// do the swaps
#pragma omp parallel for
for (int i=0; i<n*(n-1)/2; i++)
{
	int r = index[i][0];
	int c = index[i][1];
	int temp = matrix[r][c];
	matrix[r][c] = matrix[c][r];
	matrix[c][r] = temp;
}
```

This may not be the best approach in all situations but demonstrates that problems can be reformulated in order to make them more suited to efficient parallel execution.

In a distributed memory architecture, none of the above methods will work. The processors do not have access to the entire matrix. In general, we can assume that in a distributed memory architecture, the matrix will be already partitioned in a block checkerboard fashion. If we also assume that a (virtual) mesh topology is available to connect the processors, and the number of matrix elements is much larger than the number of processors, each processor will hold one block in its memory as shown in Figure 7-3. The blocks held by the processors in the main diagonal of the mesh must be swapped locally; a sequential transpose executed for the elements of the block. The elements of the other blocks, however, cannot be transposed immediately, as one processor does not have access to the elements of the reflected block held on another processor. The required extra step is to exchange blocks, then perform a local transpose within the block.



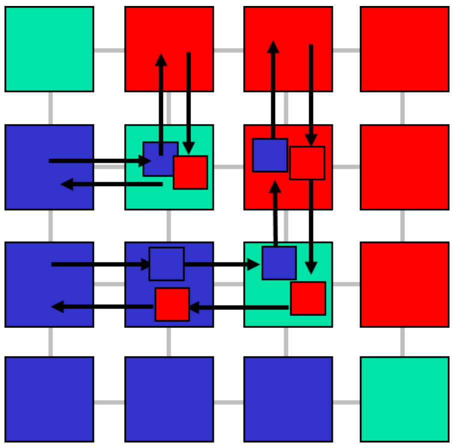


Figure 7-3 Matrix transpose on a distributed memory architecture. The arrows indicate the exchange of blocks between processors.

Figure 7-4 The physical movement of data blocks in the mesh interconnect.

If the processors are connected in a mesh topology, the blocks cannot be exchanged in one step. The blocks must be sent to the other process via the interconnect of the system. Figure 7-35 illustrates how the blocks might be sent to their destination, in most modern systems, via the mesh routing interconnection system.

Let us turn to the message passing version of this transpose algorithm. Instead of detailed implementation, only the structure of the code is shown below. We assume that data is alaready distributed and the transposed matrix does not have to be gathered to one processor – the block will stay at their processor. If data is not distributed, we need to add an extra scatter operation as an initialisation step.

Pseudo code of the parallel message-passing matrix transpose operation

```
if (my_row == my_col)        // block is on the diagonal
{
  transpose(block);          // perform seq. transpose on block
}
else
{
  if (my_col > my_row) {     // block in the upper triangle
        send(my_col, my_row, block);           // swap step 1
        receive(my_col, my_row, block);  // swap step 2
  else {
        receive(my_col, my_row, temp_block);// swap step 1
        send(my_col, my_row, data);            // swap step 2
        block = temp_block;                    // update block
  }
      transpose(block);            // perform seq. transpose on block
}
```

7.1.3 Matrix-matrix multiplication

Since we have described different versions of the parallel matrix-vector multiplication algorithm, in this chapter we move on to the matrix-matrix multiplication operation, which is the generalisation of the matrix-vector operation and shows more interesting problems from the parallel execution point of view.

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} \cdot B_{kj}$$

We have seen earlier that for the matrix-vector multiplication we need one complete row of the matrix and the complete column of the vector to calculate one element of the result vector. The matrix-matrix multiplication extends this operation as the second column elements of the result matrix **C** will need the same row of matrix **A** and the second column vector of matrix **B**. This is illustrated in Figure 7-5.

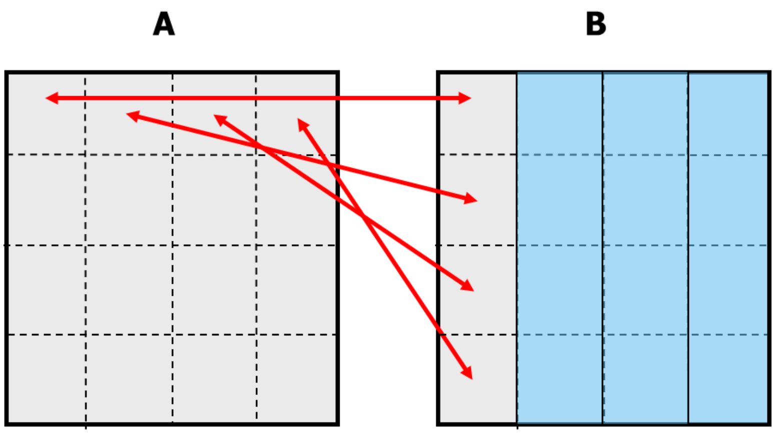


Figure 7-5 Data dependency in the matrix-matrix multiplication.

The sequential algorithm of the multiplication is the following. The first loop iterates over the rows of matrix A and C, the second loop iterates over the columns of matrix B and the inner loop visits the elements of the selected row and column, respectively to perform the multiplication. Assuming $n \times n$ square matrices, the sequential complexity of the operation is $O(n^3)$, which indicates significant execution time for large matrices.

```
for (int i = 0; i < SIZE; i++)
{   // row (A)
    for (int j = 0; j < SIZE; j++)
    {   // col(B)
        C[i][j] = 0;
```

```
            for (int k = 0; k < SIZE; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
```

Let us start again with a shared memory environment. The elements in one row of matrix **C** can be computed independently from each other. Therefore, the outer loop can be executed in parallel. Matrix A and C will be partitioned row-wise. We cannot partition matrix B since each column is require during the calculation of one row of result elements. The OpenMP implementation of this algorithm would be as follows.

```
#pragma omp parallel for schedule(static) \
          default(none) shared(A,B,C)
for (int i = 0; i < SIZE; i++)
{  // row (A)
   for (int j = 0; j < SIZE; j++)
   {   // col(B)
       C[i][j] = 0;
       for (int k = 0; k < SIZE; k++) {
           C[i][j] += A[i][k] * B[k][j];
       }
   }
}
```

The distributed memory version is a bit more complicated. If the three matrices are already distributed in a block checkerboard fashion, we need to assemble a complete row of blocks for matrix **A** and **C** and the entire matrix **B** on each processor before the multiplication can start. Not only does this require moving large amount of data but also needs large memory to store the complete matrix **B**. This results in sub-optimal performance. Note that the large matrix can be a performance problem even in the OpenMP implementation too, since the matrix might be too large to be stored in the fast cache and as a result the algorithm might become memory bound (its performance limited by the memory bandwidth).

Fortunately, there are algorithms that are more memory efficient and lend themselves to better parallel implementations. If we block partition our matrices as shown in Figure 7-6, we can rewrite the matrix multiplication in a so-called block matrix form; we treat each block as an element of the matrix formed from the blocks.

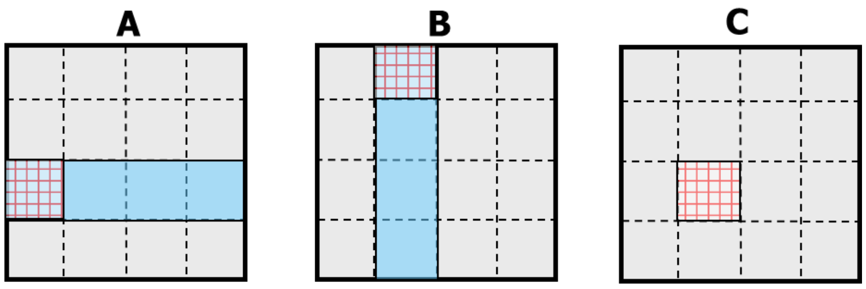


Figure 7-6 Illustration of the block matrix multiplication algorithm

Using this block partitioning we can rewrite the innermost loop of the original sequential algorithm:

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

to the following block matrix form:

```
for (k=0; k<N; k++)
   $C_{ij} = C_{ij} + A_{ik} * B_{kj}$;
```

where $\mathbf{A}_{ik}$ * $\mathbf{B}_{kj}$ represents a matrix multiplication of the blocks $\mathbf{A}_{ik}$ and $\mathbf{B}_{ki}$. The equivalence of these two forms can be checked on the following sample matrix. The two sum expressions on the right show the original loop result (top-right) and the block matrix version (bottom-right).

$$\left[\begin{array}{cc:cc} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ \hdashline a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{array}\right] \cdot \left[\begin{array}{cc:cc} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ \hdashline b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{array}\right] \begin{array}{l} a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} + a_{03}b_{30} \\ \\ \\ (a_{00}b_{00} + a_{01}b_{10}) + (a_{02}b_{20} + a_{03}b_{30}) \end{array}$$

To execute this algorithm in parallel, each processor needs to hold a row of blocks of matrix A, a column of blocks of matrix B in order to create on block as a result as shown in Figure 7-6. The structure of the parallel algorithm is as follows:

Step 0. Each processor holds blocks $\mathbf{A}_{p,q}$ and $\mathbf{B}_{p,q}$

Step 1. Each processor needs blocks $A_{p,k}$ and $B_{k,q}$ therefore execute:

A) an all-to-all broadcast of the local block of **A** to each processors in the row

B) an all-to-all broadcast of the local block of **B** to each processors in the column

Step 2. Since each processor not has the complete row of blocks from **A** and columns of blocks from **B**, perform block matrix multiplication in parallel.

This version is better than the first simple parallel version as the number of elements each processors needs to hold is reduced. However, each processor still needs a complete row and column of blocks to perform the multiplication. Cannon's [1] and Fox's [2] algorithms are two more memory efficient versions of this algorithm and are widely used in parallel numerical programs. In this section we look at Cannon's algorithm. Fox's algorithm is left as a study exercise for the reader.

Parallel matrix multiplication based on Cannon's algorithm

The key concept in Cannon's algorithm is that with a carefully designed execution schedule, each processor needs to hold only a single block of matrix **A** and **B** at any one time, hence the memory efficiency of the algorithm is optimal. Assuming *p* processors connected in a wraparound $\sqrt{p} \times \sqrt{p}$ mesh, each processor hold a block of $n/\sqrt{p} \times n/\sqrt{p}$ elements. Before the execution starts, the blocks need to be aligned to their initial position. The blocks in matrix **A** are shifted circularly to the left *i* steps, where *i* is the row index of the given block. The same is performed for matrix B but along the columns *j* steps where *j* is the column index of the block (Figure 7-7).

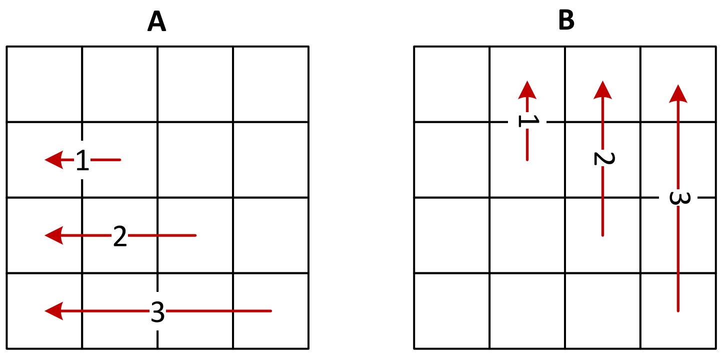


*Figure 7-7 Initial block alignment steps for matrix **A** and **B**. Blocks in A are cyclically shifted to left, blocks in B are cyclically shifted up according to their row and column positions, respectively.*

After the initial alignment, the algorithm starts with a local **A**·**B** multiplication, then each block or **A** is shifted on position to the left, and each block of **B** one position up, circularly. After $\sqrt{p}$ number of multiplication-shift steps, the two matrices are multiplied and the result is stored in matrix **C**.

The MPI implementation is the following.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#define N 1000
int main(int argc, char *argv[])
{
       MPI_Comm cannon_comm;
       MPI_Status status;
       int rank,size;
       int shift;
       int i,j,k;
       int dims[2];
       int periods[2];
       int left,right,up,down;
       double *A,*B,*C;
       double *buf,*tmp;
       double start,end;
       unsigned int iseed=0;
       int Nl;
       MPI_Init(&argc,&argv);
       MPI_Comm_rank(MPI_COMM_WORLD,&rank);
       MPI_Comm_size(MPI_COMM_WORLD,&size);
       srand(iseed);
       dims[0]=0; dims[1]=0;
       periods[0]=1; periods[1]=1;
       MPI_Dims_create(size,2,dims);
       if(dims[0]!=dims[1]) {
              if(rank==0) printf("The number of processors must be a square.\n");
              MPI_Finalize();
              return 0;
       }
       Nl=N/dims[0];
       A=(double*)malloc(Nl*Nl*sizeof(double));
       B=(double*)malloc(Nl*Nl*sizeof(double));
       buf=(double*)malloc(Nl*Nl*sizeof(double));
       C=(double*)calloc(Nl*Nl,sizeof(double));
       for(i=0;i<Nl;i++)
```

```
            for(j=0;j<Nl;j++) {
                  A[i*Nl+j]=5-(int)( 10.0 * rand() / ( RAND_MAX + 1.0 ) );
                  B[i*Nl+j]=5-(int)( 10.0 * rand() / ( RAND_MAX + 1.0 ) );
                  C[i*Nl+j]=0.0;
            }
            MPI_Cart_create(MPI_COMM_WORLD,2,dims,periods,1,&cannon_comm);
            MPI_Cart_shift(cannon_comm,0,1,&left,&right);
            MPI_Cart_shift(cannon_comm,1,1,&up,&down);
            start=MPI_Wtime();
            for(shift=0;shift<dims[0];shift++) {
                  // Matrix multiplication
                  for(i=0;i<Nl;i++)
                        for(k=0;k<Nl;k++)
                              for(j=0;j<Nl;j++)
                                    C[i*Nl+j]+=A[i*Nl+k]*B[k*Nl+j];
                  if(shift==dims[0]-1) break;
                  // Communication
                  MPI_Sendrecv(A,Nl*Nl,MPI_DOUBLE,left,1,
                        buf,Nl*Nl,MPI_DOUBLE,right,1,cannon_comm,&status);
                  tmp=buf; buf=A; A=tmp;
                  MPI_Sendrecv(B,Nl*Nl,MPI_DOUBLE,up,2,
                        buf,Nl*Nl,MPI_DOUBLE,down,2,cannon_comm,&status);
                  tmp=buf; buf=B; B=tmp;
            }
            MPI_Barrier(cannon_comm);
            end=MPI_Wtime();
            if(rank==0) printf("Time: %.4fs\n",end-start);
            free(A); free(B); free(buf); free(C);
            MPI_Finalize();
            return 0;
}
```

The presented source code can be found along with a comparison of various other matrix multiplication algorithms and their performance on the IBM BlueGene computer in [3]. Note the use of the Cartesian virtual topology and its use for shifting, the one-step MPI_Sendrecv() routine that sends a block and receives the next block into the same buffer, and the use of the barrier synchronisation after each shift step.

7.2 SOLVING SYSTEMS OF LINEAR EQUATIONS

Systems of linear equations are central to many scientific and engineering problem. After the matrix manipulation algorithms, solving linear equation systems is probably the most widely used operation in numerical computing.

We assume that we have a system of liner equations in the following form.

$$
\begin{aligned}
&a_{0,0}x_0 \quad + a_{0,1}x_1 \quad + \cdots + a_{0,n-1}x_{n-1} \quad = b_0 \\
&a_{1,0}x_0 \quad + a_{1,1}x_1 \quad + \cdots + a_{1,n-1}x_{n-1} \quad = b_1 \\
&\vdots \\
&a_{n-1,0}x_0 + a_{n-1,1}x_1 \quad + \cdots + a_{n-1,n-1}x_{n-1} = b_{n-1}
\end{aligned}
$$

We can express this in matrix form, such as

$$\mathbf{Ax} = \mathbf{b}$$

We further assume that the matrix **A** is dense and hence the equations can be solved by direct, matrix manipulation methods. Due to space considerations, solution techniques of sparse systems is not covered in this book.

The simplest method for solving our equations is the Gauss elimination. Our goal in the Gauss elimination is to convert the system of equations into an upper triangular representation, in which the elements of the lower triangle are zeros and the elements of the main diagonal are 1s. Once this representation is achieved, the system of equations can be solved using back-substitution. The method relies on the property of linear equations that any row can be replaced by the same row added to another row that is multiplied by a constant.

The method works in row-order starting with the top row. For each selected row *i*, all rows below are replaced by a modified row. The new value for row *j* is: row *j* + (row *i*) ($-a_{ji}/a_{ii}$). Since

$$a_{j,i} = a_{j,i} + a_{i,i}\left(\frac{-a_{j,i}}{a_{i,i}}\right) = 0$$

this results in zero elements in column *i* below row *i*, as indicated in Figure 7-13.

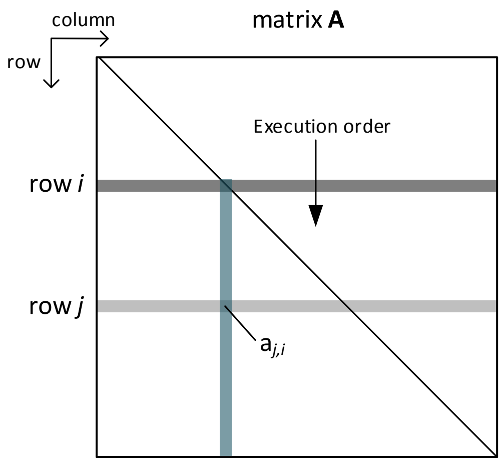


Figure 7-8 Graphical illustration of the execution of the elimination step. The blue column indicates the actual column whose elements are set to zero in a given iteration i.

The complete sequential algorithm is as follows. Each iteration starts with dividing each element in the row by a_{ii} to ensure that the main diagonal element becomes 1. Next we start the elimination step as described above. The elimination step also modifies the vector **b** to keep our data consistent.

```
for i=0 to n-1
   for j=0 to n-1
      $a_{ij}$:= $a_{ij}/a_{ii}$       {division step}
   $y_i$:= $b_i/a_{ii}$
   $a_{ii}$:= 1
   for j=i+1 to n-1
      for k=i+1 to n-1
         $a_{jk}$:= $a_{jk}$ - $a_{ji}*a_{ik}$  {elimination step}
      $b_j$:= $b_j$ - $a_{ji}*y_i$
      $a_{ji}$:= 0
   endfor
endfor
```

The Gauss elimination has quite a few known deficiencies. It is sensitive to the elements due to the division operation and can easily become numerically instable. There are techniques that improve the stability of the algorithm such as partial pivoting. Since these techniques are not strictly related to the parallelisation issues, we do not discuss them in this book. Interested readers are encouraged to study the literature for more advanced solution methods.

Where is the potential for parallelism in this algorithm? The outer loop cannot be changed as the method is based on transforming the rows in sequence. The elimination step, however, can be executed in parallel for each row *j*.

In a shared memory implementation, we do not need to change much to make the algorithm run in parallel. Inserting a *parallel for* OpenMP directive in front of the main elimination loop (outlined in blue below) will do the job. We need to be careful about the work scheduling, however, since the workload in the algorithm decreases continuously as the iterations proceed. This is due to the fact that as the program reaches higher row indices, the number of elements right of and below the diagonal are becoming smaller and smaller. Using the default static work schedule policy would allocate an equal number of rows to the processors, some receiving many, while some receiving very few elements. The dynamic policy will balance the load of the processors more evenly.

Gauss elimination: OpenMP version pseudo code

```
for i=0 to n-1
   for j=0 to n-1
      $a_{ij}$:= $a_{ij}/a_{ii}$       {division step}
   $y_i$:= $b_i/a_{ii}$
   $a_{ii}$:= 1
#pragma omp parallel for schedule(dynamic, 1)
   for j=i+1 to n-1
      for k=i+1 to n-1
         $a_{jk}$:= $a_{jk}$ - $a_{ji}*a_{ik}$  {elimination step}
      $b_j$:= $b_j$ - $a_{ji}*y_i$
      $a_{ji}$:= 0
   endfor
endfor
```

The message-passing parallel implementation is similar in concept but we need to focus on the data partitioning and movement issues. Since the algorithm is complete row oriented, it makes sense to use striped partitioning Assuming for a minute that one processor holds one row locally, the processors performing the elimination step will all need a copy of row *i*. This will require a broadcast step in the main row iteration loop.

Gauss elimination: MPI version pseudo code

```
for i=0 to n-1
   for j=0 to n-1
      $a_{ij}$:= $a_{ij}$/$a_{ii}$                {division step}
   $y_i$:= $b_i$/$a_{ii}$
   $a_{ii}$:= 1
   root = i;                      {row }
   MPI_Bcast(...,root,...)        {broadcast row i}
   for j=i+1 to n-1
      for k=i+1 to n-1
         $a_{jk}$:= $a_{jk}$ - $a_{ji}$*$a_{ik}$      {elimination step}
      $b_j$:= $b_j$ - $a_{ji}$*$y_i$
      $a_{ji}$:= 0
   endfor
endfor
```

As we mentioned above, this algorithm assumes that one processor holds one row, therefore the root of the broadcast operation is always the processor with rank = i. If we work with large matrices, the number of processors in the system is most likely to be less than the number of rows of the matrix. If we use striped block partitioning, each processor will hold an equal number of rows. The root rank for the broadcast in this case is calculated as $\lfloor i/(n/p) \rfloor$. The block distribution will result in inefficient load distribution; most processors will do nothing as the row index is increasing. A better approach may be to use a cyclic striped partitioning, in which scheme the rows are allocated to the different processors in a sequence with wraparound. In this case the rank of the processor that holds and broadcasts row *i* is $i \bmod (n/p)$. More details of the parallel MPI version of the Gaussian elimination can be found in [4].

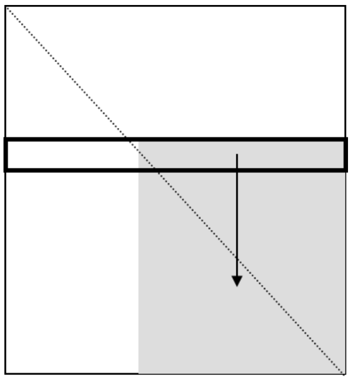


Figure 7-9 The shrinking active area of the matrix represents decreasing workload as the algorithm proceeds.

7.3 Solving partial differential equations

Solving partial differential equations (PDE) is a frequently occurring problem in many fields of science and engineering. In many situations, the only possible solution is numerical approximation. One important type of PDE is the Laplace equation, defined as:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

Without detailing the theoretical background (for details see [5]), we can say that these equations are solved using the finite difference method. The problem space (in the above case two-dimensional) is discretised into a fine mesh of points placed at a regular distance apart as shown in Figure 7-10. If the distance between the points is small enough, the second derivative can be approximated with the following finite difference formula:

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{1}{d^2}[f(x+d,y) - 2f(x,y) + f(x-d,y)]$$
$$\frac{\partial^2 f}{\partial y^2} \approx \frac{1}{d^2}[f(x,y+d) - 2f(x,y) + f(x,y-d)]$$

Substituting this into the Laplace equation, after rearrangement and setting $d = 1$, we get the following formula:

$$f(x,y) = \frac{1}{4}[f(x-1,y) + f(x,y-1) + f(x+1,y) + f(x,y+1)]$$

This states that the function at point *x*, *y* can be computed using only its four neighbours. By re-phrasing it as an iterative formula:

$$f^k(x,y) = \tfrac{1}{4}[f^{k-1}(x-1,y) + f^{k-1}(x,y-1) + f^{k-1}(x+1,y) + f^{k-1}(x,y+1)] \qquad \text{Eq.1}$$

where $f^k(x,y)$ is the value obtained in the *k*th iteration and the value $f^{k-1}(x,y)$ is obtained in the (k-1)th iteration. This iterative formula allows us to solve this equation in an iterative manner instead of a direct matrix/vector based method we saw in the previous section. An iterative solution is preferable if the matrices used in the solution would be very large and potentially sparse, having only very few non-zero elements. Also, the direct method has a complexity $O(n^2)$, whereas the iterative solution can stop after the convergence criteria is reached, consequently the iterative method can provide a solution much faster. The sequential algorithm for this iteration is:

```
for(int iter=0; iter < limit; iter++)
```

```
{
    for(int i=1; i < n; i++)
        for(j=1; j < n; j++)
            g[i][j] = 0.25 * (h[i-1][j] + h[i][j-1] + h[i+1][j] + h[i][i+1]);
    for(int i=1; i < n; i++)
        for(j=1; j < n; j++)
            h[i][j] = g[i][j]; // update the function points
}
```

This iteration must be repeated until our approximation of the solution is good enough or reached the limit on the number of iterations. The simplest measure of goodness of the approximation is how much a value is changing from one iteration to the other. If the change for each value is below a pre-set tolerance value ε, we stop the iteration. More formally, the convergence criteria is

$$|x_i^k - x_i^{k-1}| < \varepsilon$$

where $x_i^k$is the value of element x_i in the kth iteration, and similarly x_i^{k-1} is the value in the (k-1)th iteration.

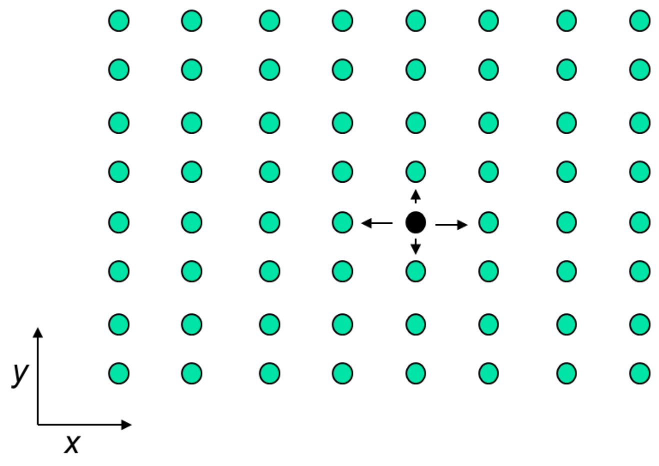


Figure 7-10 The discretised parameter space for two variables x and y.

Unfortunately, the convergence rate of the finite difference expression Eq.1 is very slow, therefore in practice a faster method, the Gauss-Seidel overrelaxation is used [5]. Since in discussing strategies for creating a parallel version the convergence rate is secondary, we use the original version in our discussion.

The shared memory parallel implementation using OpenMP is relatively straightforward. Since each new value is computed independently from the others, a striped block partitioning can be used within the convergence loop. The new value computation will be executed in one parallel region, the value update in another one.

```
for(int iter=0; iter < limit; iter++)
{
    #pragma omp parallel for
    for(int i=1; i < n; i++)
        for(j=1; j < n; j++)
            g[i][j] = 0.25 * (h[i-1][j] + h[i][j-1] + h[i+1][j] + h[i][i+1]);
    #pragma omp parallel for
    for(int i=1; i < n; i++)
        for(j=1; j < n; j++)
            h[i][j] = g[i][j]; // update the function points
}
```

The message passing version is trickier as one new value calculation involves four other neighbour points. When the point mesh is distributed, some points will not have access to their previous neighbours (previous in the sequential implementation sense) as shown in Figure 7-11. The solution to this problem to introduce an extra so-called ghost column (column-wise striped partitioning) or row (row-wise striped partitioning) that holds the former neighbour points now stored at a neighbour processor locally. These are illustrated by grey points in Figure 7-11.

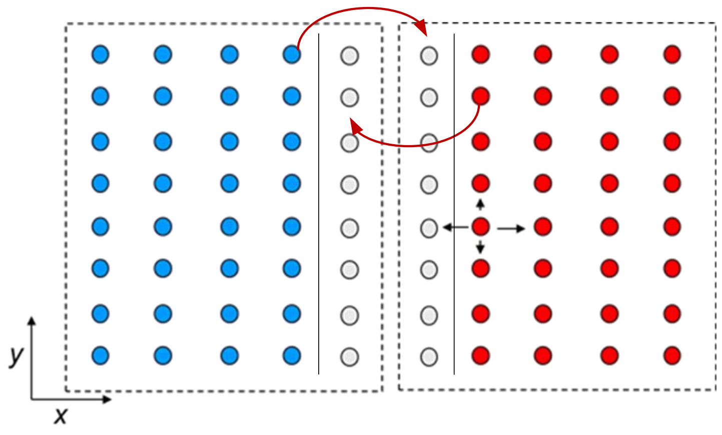


Figure 7-11   The discretised mesh partitioned into two parts. Note the need for ghost columns that must hold the boundary values of the neighbouring processor for the four-point calculation to be executed correctly.

Parallel MPI pseudo code

```
//initialisation step
//partition original matrix holding the mesh values and distribute segments
//using MPI_Scatter ensuring that there is space allocated for ghost rows/columns
//this version uses row-wise partitioning
for(int iter=0; iter < limit; iter++)
{
    //update ghost rows
    if (myid % 2 = 0)
    {
        // send last row of segment to next proc.
        MPI_Send(&h[myid * segment_height-1], n, MPI_DOUBLE, myid + 1, ...);
        // receive ghost row from next proc.
        MPI_Recv(&h[myid * segment_height], n, MPI_DOUBLE, myid + 1, ...);
    }
    else
    {
        // receive ghost row from previous proc.
        MPI_Recv(&h[0], n, MPI_DOUBLE, myid - 1, ...);
        // send first row to previous proc.
        MPI_Send(&h[1], n, MPI_DOUBLE, myid - 1, ...);
    }
    // perform compation within the segment
    for(int i=1; i < segment_height; i++)
        for(j=1; j < n; j++)
            g[i][j] = 0.25 * (h[i-1][j] + h[i][j-1] + h[i+1][j] + h[i][i+1]);
    for(int i=1; i < segment_height; i++)
        for(j=1; j < n; j++)
            h[i][j] = g[i][j]; // update the function points
}
```

7.4 Sorting

One of the most common operations in computing is sorting. There are numerous algorithms for sorting data. In this section we will only concentrate on one specific algorithm and its parallel version, namely the *odd-even transposition* sort.

This algorithm belongs to the class of *comparison-based* sorting algorithms, which perform sorting by repeatedly comparing elements and changing them if their order is not as required. Other algorithms in this class are e.g. Selection, Insertion, Bubble, Quicksort, Merge sort. The difference among these algorithms is the number of operations required for the complete sort. Some have a better lower bound complexity, some performs better on average, and some are more suited to large data sizes.

The complexity of the best sequential sorting methods is $O(n \log n)$, therefore the optimal performance we can expect from a parallel implementation is $O(n \log n)/p$. Theoretically, using $p = n$ processors we may reach $O(\log n)$, but it is a very difficult task in practice [6].

Before examining the techniques for creating a parallel sort implementation, let us revisit how comparison based sequential sorting works. The fundamental operation in comparison sort is the **compare-exchange** step. Two elements, A and B, are compared and exchanged if their order is not correct. This is illustrated by the following code sample. If we want numbers in ascending order and A is larger than B, we need to swap, otherwise we do nothing.

```
if (A > B)
{
    temp = A;
    A = B;
    B = temp;
}
```

Let us wrap this operation into a routine and call is *compare_exchange*(a, b). Using this function we can build sequential sort algorithms. One of the mostly taught but worst performer algorithms is bubble sort. The algorithm sweeps through the sequence on numbers starting with comparing the first two. If needed, they are swapped. Then the second and third are compared and if necessary, swapped. To reach the end of the sequence, we need $n-1$ compare-exchange steps. This round moves the largest number to the end of the sequence. In the next round we only need $n-2$ comparison steps as the largest one is already at the end of the list. Then we only need $n-3$, and so on until we are left with two elements.

Bubble sort: sequential algorithm

```
bubble_sort(n)
begin
   for i:= n-1 downto 1 do
```

```
        for j:= 1 to i do
            compare_exchange($a_j$, $a_{j+1}$);
end
```

The algorithm is named after the bubbling effect of how the large numbers move to the end of the sequence. The complexity of Bubble sort is $O(n^2)$, which is far way from the optimal $O(n \log n)$. A further problem is that Bubble sort in its original form is an inherently sequential operation. Each iteration depends on the compare-exchange operation of the previous one. There are variants of this algorithm that will be better suited to parallel implementation.

In the odd-even transposition algorithm, the element pairs are compared in alternating *even* and *odd* phases. In the even phase, elements with even indices are compared with their right neighbours and changed if necessary. During the odd phases, the odd indexed elements are compared to their right neighbours. The algorithm requires *n* phases to complete and each phase requires *n*/2 compare-exchange steps.

Odd-even transposition sort: sequential algorithm

```
odd_even_sort(n)
begin
    for i:= 1 to n do
        if (i is odd) then
            for j:= 0 to n/2-1 do
                compare_exchange($a_{2j+1}$, $a_{2j+2}$);
        else
            for j:= 1 to n/2-1 do
                compare_exchange($a_{2j}$, $a_{2j+1}$);
    endfor
end
```

This version is more suitable for parallel implementation than the Bubble sort. The comparisons in one phase (odd or even) can be executed in parallel as shown in the following figure.

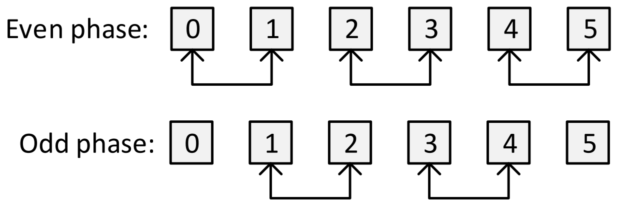


Figure 7-12 Executing pairwise compare-exchange operations in parallel.

The main question in the parallel implementation is how to map the elements onto the processors. If we map a single element to a processor, as in Figure 7-12, the number of processors *p* might become very large and should change as the number of elements changes. A more practical approach is to use $p < n$ processors but in this case the compare-exchange in its original form cannot be used; we have more than two elements in the comparison. For the $p < n$ case, we introduce a modified operation, the *compare-and-split*. In the compare-and-split step each processor merges the two sub sequences that needs to be compared, sorts it internally, then splits it in the middle to create one sub sequence from the smaller elements and one sub sequence from the larger elements.

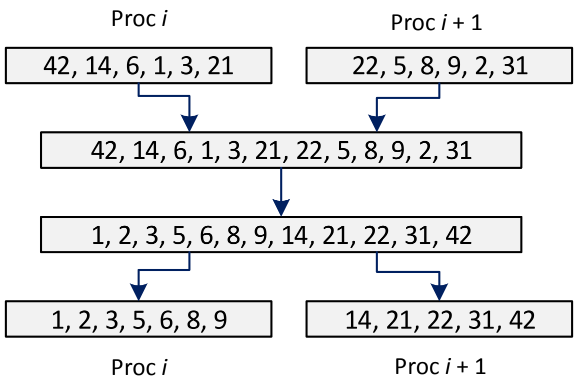


Figure 7-13 Illustration of the compare-and-split operation

In order to define the parallel algorithm, we introduce two routines. The `compare_exchange_min` will perform a *merge-sort-split* sequence and keep the minimum half of the sorted sequence. The input sub sequences are one sequence from the designated left or right neighbour and its locally stored sequence on elements. `compare_exchange_max` will perform the same operations but keep the maximum half of the elements.

The parallel algorithm

```
odd_even_sort_par(n)
   begin
      id := proc id
      for i:= 1 to n do           //perform n phases
         if (i is odd) then       //odd phase
            if (id is odd) then  //I'm on the left, compare with right proc
               compare_exchange_min(id+1);
            else                 // I'm on the right, compare with left proc
            compare_exchange_max(id-1);
        else                    //even phase
          if (id is even) then // I’m on the left, compare with right proc
            compare_exchange_min(id+1);
          else                 // I’m on the right, compare with left proc
            compare_exchange_max(id-1);
      endfor
    end
```

When implementing the compare-exchange-min/max routines, the internal sort can be performed with any suitable sequential sorting algorithm. One of the obvious choices is Quicksort since in many libraries this is the available sort function.

REFERENCES

[1] L. E. Cannon, “A cellular computer to implement the kalman filter algorithm,” Jan. 1969.

[2] G. . Fox, S. . Otto, and A. J. . Hey, “Matrix algorithms on a hypercube I: Matrix multiplication,” *Parallel Comput.*, vol. 4, no. 1, pp. 17–31, Feb. 1987.

[3] M. Cytowski, “Matrix Multiplication on Blue Gene/P.”

[4] S. F. McGinn and R. E. Shaw, “Parallel Gaussian elimination using OpenMP and MPI,” in *Proceedings 16th Annual International Symposium on High Performance Computing Systems and Applications*, 2002, pp. 169–173.

[5] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and distributed computation : numerical methods*. London :: Prentice-Hall International (UK),, 1989, p. 715.

[6] B. Wilkinson and C. M. Allen, *Parallel programming : techniques and applications using networked workstations and parallel computers*. Upper Saddle River, N.J. :: Prentice Hall,, 1999, p. 431.