Scientific Computing
on
High Performance Computers

L. Eldén
Department of Mathematics
Linköping University

H. Park and Y. Saad
Department of Computer Science
University of Minnesota

Contents

Chapter 1

Basic Concepts

This chapter gives an overview of the basic concepts in vector and parallel numerical algorithms. The goal of this chapter is to introduce the main ideas of vector and parallel algorithms with little attention given to the underlying architecture. We have abstracted the architecture from the algorithm when introducing these basic concepts to underscore the fact that these are general notions valid for all implementations.

1.1 What is a High Performance Computer?

In this book we will use the term **High Performance Computer** to denote any advanced architecture computer with the following features:

- parallelism on different levels
- two or more levels of memory with different access times, a so called memory hierarchy

Within this class we have different kinds of parallel computers, and several advanced workstations. Although these architectural features have been introduces to make the computers execute fast, it is not computational speed as such that defines the notion of high performance computer. In fact, a fast parallel computer may execute more than 100 times faster than a workstation (see Tables 1.1 and 1.2), and still they should both be called high performance computers.

We will refer to non-high-performance-computers as **conventional** or **sequential computers**, although such computers may have some parallel features (especially on a low level of the hardware), but to a lesser extent than high performance computers.

To illustrate the aspect of speed, we cite the following performance figures from a commonly used computer benchmark, the LINPACK benchmark[7]. In that test the performance of computers is measured from the execution times for solving two linear systems of equations: the first is of dimension 100 and is solved using a subroutine from the LINPACK library[8]. The second is of dimension 1000 and any code can be used. A third benchmark is to attain the highest possible speed on the computer by solving a large

enough linear system. In Tables 1.1 and 1.2 we give timings in Mflops (million floating point operations per second) for some parallel computers and workstations.

Computer	$n = 100$	$n = 1000$	Peak performance
Cray T932 (32 proc.)		29360	57600
NEC SX-4/32 (32 proc.)		31060	64000
NEC SX-4/1 (1 proc.)	578	1944	2000
IBM RS/6K 44P-270	426	1109	1500
SUN UltraSPARC II (1 proc.)	154	461	672

Table 1.1: LINPACK Benchmark for various computers, January 18, 2001.

1.2 Why High Performance Computing?

High performance computers are designed to be very fast and are intended for problems that would otherwise be intractable. There has been increasing demand from scientific computing applications, where problems are becoming more and more complex and the prospect of increasing computational power is in turn spurring the demand for faster machines to solve these harder problems. We will give two examples of applications, where the use of supercomputers is essential. In the description of the examples we will use the terms Mflops and Gflops , which are measures for the speed of fast computers. 1 Mflops and 1 Gflops are the same as 10^6 and 10^9 floating point operations per second, respectively.

The first example is the simulation of car crashes, where a car hits a wall. The model of the car has approximately 20,000 elements, with 6 degrees of freedom (unknowns) for each element, i.e., 120,000 unknowns altogether. The crash is simulated during 120ms, and 150,000 time steps are taken. In each time step around 100 floating point operations (flops) are needed per unknown. This means that the total computation requires $1.2 \cdot 10^5 \cdot 1.5 \cdot 10^5 \cdot 10^2 \approx 2 \cdot 10^{12}$ flops. On a computer with a speed of 1 Mflops, this would take $2 \cdot 10^6$ seconds or 25 days approximately. This is too time consuming in a product development stage, when it is necessary to evaluate several alternative constructions. On

Computer	Number of processors	R_{max} Gflops/s	N_{max} order	R_{peak} Gflops/s
ASCI White (IBM)	7424	4938	430000	11136
Cray T3E	1080	891	259200	1296
SUN Ultra HPC10000	256	100	80000	128
SGI Origin 2000	256	98	81920	140

Table 1.2: The solution of a linear system of order N_{max} on some highly parallel computers (Linpack Benchmark, January 18, 2001). R_{max} is the attained speed.

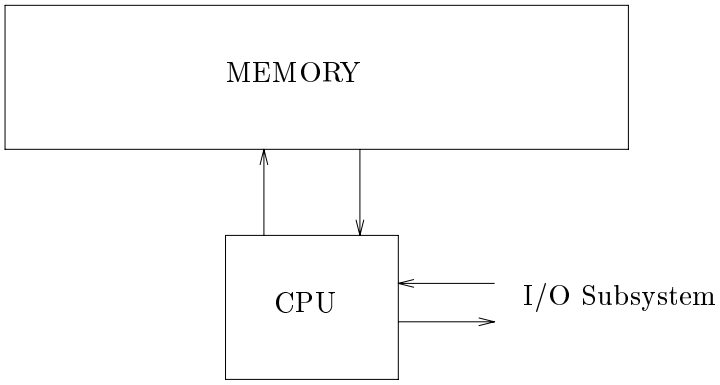


Figure 1.1: The von Neumann model.

the other hand, with a computer that can run at 10 Gflops, the same computation takes about 3.5 minutes, which is acceptable.

The second example is concerned with the flow around a space-shuttle. In this computation there are around 10^6 grid points and 5 unknowns per point. 10^4 time steps are taken, and for each unknown and step, 10^3 flops are needed. This adds up to $5 \cdot 10^{13}$ flops altogether, and from the previous example we see that this needs a computer with a speed of at least 1 Gflops to be feasible.

1.3 Concept of Parallel Computer

Although the concept of parallel computing was present in the pre-electronic computing era of the 19th century, it was not until the recent years that the exploitation of this concept has taken place. Babbage and others recognized parallel processing as a means for speeding up the multiplication of two numbers [3] in his differential engine. The von Neumann model of computing, which prevailed at the beginning of high speed computing, lead to architectures that were intrinsically sequential. The von Neumann model is illustrated in Figure 1.1.

The basic idea in this model is that data and instructions are fetched one at a time or a few at a time, and then executed. A typical execution sequence would be as follows:

- fetch next instruction and decode it;
- calculate addresses of operands;
- fetch operands;
- execute instruction;
- return resulting operand to memory.

The major limitation in this model is the I/O of data to and from memory. Even if we assume that fast computation can be achieved, the bus to and from memory can cause a bottleneck. In fact this is referred to as the von Neumann bottleneck . The early versions of 'supercomputers', such as the CDC 7600, included some parallel processing, mainly in the form of duplicating a few functional units. The first major improvements in von Neumann model came in the early supercomputers in which memories were organized into banks so as to allow a form of parallel access to and from memory.

The recent major turn to parallel computing is largely motivated by the limitations inherent in the von Neumann model. Parallelizm can be achieved either within a single processor via vectorization or by increasing the number of processors in a system. In a parallel computer, which is a system with multiple processors, different processors share the computations involved in the solution of a problem. To this end, the computation must be decomposed and broken up into tasks, which can be performed simultaneously by the different processors; and organized co-operation among the processors must be established by means of synchronization or data exchange. The selection of an algorithm, its decomposition into separate computational tasks and their subsequent assignment to particular processors, as well as the physical channels and protocols by which the processors communicate are among the many factors leading to a multitude of parallel implementations for any one problem.

The above decisions are made more difficult by the need (or perhaps absence) of adequate performance measures. Even if a certain multiprocessor machine is already specified, one still faces the problem of having to decompose a particular algorithm into tasks with the objective of reducing execution time to a minimum, and achieving a balanced work-load among all processors. Before this, however, reliable criteria for evaluating and comparing the performance of different implementations of an algorithm on a particular machine could be crucial tools. Naturally, it is also important to be able to compare implementations of different algorithms on *one* machine, as well as implementations of different algorithms on *different* machines. These issues are far from being resolved. One of the reasons is that a fair assessment of two architectures must be based on the availability of adequate hardware as well as software. Yet, due to the absence of systematic design techniques, the development of software is and will continue to be lagging behind hardware development.

1.4 Performance Measurements

1.4.1 Speedup and Efficiency

In ideal situations, one should expect to gain a factor of p in time when using p processors to solve a given problem. The definition of speed-up is rather straightforward. If T_1 is the time required to execute the algorithm on one processor, and T_p the time to execute it on p processors, then the speed-up is

$$S_p \equiv \frac{T_1}{T_p}. \tag{1.1}$$

This is an intuitive notion but it contains a rather serious limitation. Often, parallel algorithms are not the best ones on one-processor machines, since they contain additional operations that may not be needed in a one processor machine. These could be additional arithmetic or simply communication and synchronization operations. For fairness, we should be comparing the *best possible algorithm* on a one-processor machine, say algorithm A, versus the specific parallel algorithm, on a p-processor machine,

$$S_p^A \equiv \frac{\text{Time for a serial algorithm A}}{T_p}, \tag{1.2}$$

The trouble is that in many instances one does not know what the best sequential algorithm is and there might be many arguments for and against this modified definition. An alternative is to still consider the same algorithm on a one processor machine as on a multi-procesor machine but to take into account the overhead due to a parallel execution, i.e., to emulate the parallel algorithm on a one processor machine. This may become cumbersome for realistic problems.

In this book we will refer to the above speed-up defined by (1.1) as the *theoretical speed-up* and to the one defined by (1.2) as the *actual speed-up* or *relative speed-up with respect to* a given algorithm (A).

The efficiency is simply the speed-up divided by the number of processors. We can similarly define two such notions,

$$E_p = \frac{S_p}{p} \quad \text{and} \quad E_p^A = \frac{S_p^A}{p}\ . \tag{1.3}$$

Note that $E_p \leq 1$ whereas E_p^A is basically arbitrary. Ideally we would like to always have $E_p = 1$, i.e., to have an optimal return for using more processors. Unfortunately, this is almost never reached for realistic algorithms.

On a vector processor, the speed-up (or efficiency) can be defined analogously as

$$S_v = \frac{T_1}{T_v}$$

where T_v and T_1 are the times required to execute the algorithms on a processor with and without the vectorization capability.

1.4.2 Amdahl's Law for Parallel Computing

Before we actually run a program to answer the question of whether it pays to run it on a parallel computer (in the sense that it utilizes the hardware efficiently), it will be useful to have a theoretical tool to predict the speed-up and efficiency. Therefore, it is interesting to study the performance of a code on a vector or parallel computer, where only a certain fraction can be vectorized or parallelized. In addition to the usual overhead due to communication and synchronization operations in parallel algorithms, one often finds algorithms that have a portion that is intrinsically sequential.

Let us assume that the fraction of the perfectly parallelizable computation in an algorithm is f_p and that the rest, i.e., $1 - f_p$ is serial. If the total execution time of this

algorithm on one processor is T_1, then the execution time for the intrinsically sequential part is $(1-f_p)T_1$ and the rest is f_pT_1. On p processors, the first part cannot be parallelized so its execution time will be the same, whereas the second part can be parallelized by a factor of p, leading to the parallel execution time

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

Therefore, the speed-up is

$$S_p \leq \frac{1}{1 - f_p + \frac{f_p}{p}}. \tag{1.4}$$

Note that this means that for any algorithm and any number of processors,

$$S_p \leq \frac{1}{(1 - f_p)}.$$

Formula (1.4) is referred to as *Amdahl's Law for Parallel Computing*. The main consequence of Amdahl's law is that if a substantial portion, say 90%, of the computation in an algorithm is scalar then, eventually it does not pay to throw in more processing power, since the advantage becomes very little. The upper bound of the speed-up that can be reached is $1/(1-f_p)$ and the efficiency will decrease asymptotically as $1/((1-f_p)p)$. Thus, if $f_p = 99\%$ then we cannot achieve speed-up of the parallel implementation of the algorithm by more than a factor 100, no matter how many processors are used.

1.4.3 Amdahl's Law for Vector Computing

Many high performance computers can execute arithmetic operations in two modes: scalar mode and vector mode. In Chapter 2 we will describe the idea of vector operations in more detail; for the present discussion it is sufficient to point out that vector operations are executed much faster than scalar operations.

Assume that we have a vector processor and that operations in scalar mode and in vector mode take t_s and t_v (in some unit), respectively. Then the peak performance of this computer is

$$r_\infty \equiv \frac{1}{t_v}.$$

Further assume that the fraction f_v of the total number of operations in the code can run in vector mode. Then the total time to execute the code is

$$T = N[(1 - f_v)t_s + f_v t_v],$$

where N is the number of operations performed in the code. The average time per operation is

$$t_f = [(1 - f_v)t_s + f_v t_v],$$

and the performance of the computer on this code is

$$r_f = \frac{1}{(1 - f_v)t_s + f_v t_v}.$$

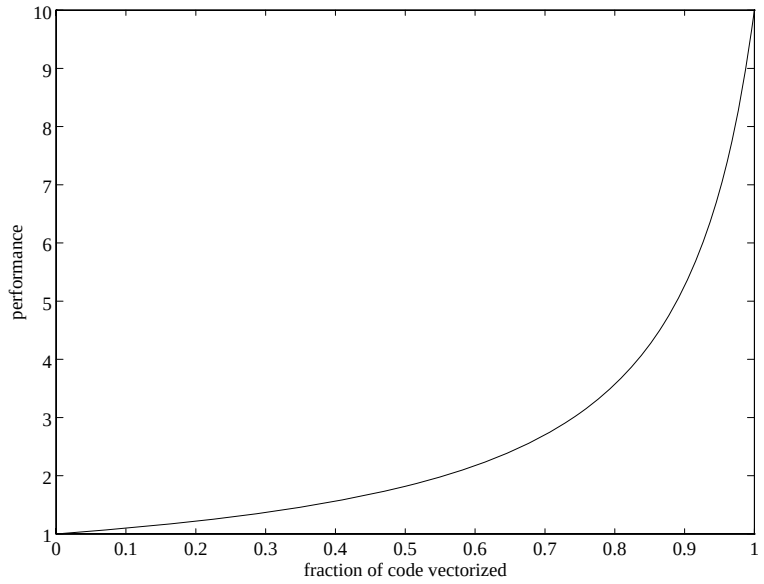


Figure 1.2: Amdahl's law for Vector Computing. We have assumed $t_s = 10t_v$, and scaled so that peak performance r_∞ is equal to 10.

This is *Amdahl's Law for Vector Computing*. In Figure 1.2 we plot r_f as a function of f_v. We have assumed that vector operations are 10 times faster than scalar operations. It is seen that only for f_v close to 1, does the performance get in the neighborhood of r_∞.

Assume that we have a code, which is vectorizable to 80%. With the values of the parameters in Figure 1.2, we then get a performance of only 36% of peak performance. Thus, Amdahl's law gives a rather pessimistic picture of the usefulness of a vector computer.

Amdahl's law is a little misleading in many situations. When one is evaluating a new computer with vector instructions, then one often wants to run bigger problems than are possible on the presently available computer. Also, often the fraction of the code that is vectorizable depends on the problem size.

Assume that the number of operations for a certain program depends on the problem size n in the following way:

$$A(n) = an^3 + bn^2,$$

where the first term represents the vectorizable part, and the second the non-vectorizable. For example, we can assume this form for the Gaussian elimination with partial pivoting, where the transformations of the matrix elements require $O(n^3)$ operations and can be vectorized well. The pivot search, on the other hand, takes $O(n^2)$ operations and is essentially sequential. Suppose for a certain value of n 80% of the code is vectorizable (i.e., $an^3/(an^3+bn^2) = 0.8$). This gives the relation, $an^3 = 4bn^2$. If we consider a problem

that is 10 times larger, then we get

$$A(10n) = 10^3 an^3 + 10^2 bn^2,$$

and the fraction vectorizable code is

$$\frac{10^3 an^3}{10^3 an^3 + 10^2 bn^2} = \frac{1000}{1000 + 25} \approx 0.976.$$

With the same parameters as in Figure 1.2 we now get 82% of peak performance.

1.4.4 Speed of a Computer

Earlier we have introduced the measures of speed Mflops and Gflops. Now we will discuss the concept of cycle time, and its relation to the measures of speed. Time in a digital computer should be considered to be *discrete*: all events take place at distinct points in time, and the **cycle time** is the constant time between these points. The fastest a sequential (i.e. non-parallel) computer can execute is one instruction per clock cycle. So, if the cycle time is 4 ns (1 ns is 10^{-9} second), then the maximum speed is $1/(4{\cdot}10^{-9}) = 2.5{\cdot}10^8$ instructions per second, i.e., 250 Mips (1 Mips is 1 million instructions per second).

Similarly, if the floating point arithmetic units of a computer can deliver one result per clock cycle, then the maximum theoretical speed for floating point operations is 1 over the cycle time. Thus, a computer with a cycle time of 4 ns can have a maximum theoretical speed of 250 Mflops under the above assumption. Later we will see that it is possible to double that figure (or increase it by a larger factor) by introducing more parallelism.

It is interesting to note that the cycle times of high performance computers has not been decreased very much, since the first supercomputer, the Cray-1, was introduced in 1976. The Cray-1 had a cycle time of 12.5 ns. The presently fastest computers have cycle times of the order of a couple of nanoseconds. However, present day machines are more than 100 times faster than the Cray-1; it is obvious then that this increase in speed is explained by a higher degree of parallelism than in the Cray-1 (cf. also the discussion in Section 1.2).

Chapter 2

Vector and Parallel Architectures

The goal of this chapter is to introduce the important architectural concepts in modern high performance computers. The emphasis is on the essential architectural concepts for designing efficient numerical algorithms on vector and parallel computers rather than on hardware-related details.

2.1 Overview of Vector and Parallel Architectures

We first present the three basic types of architectures of high performance computers that will be discussed in this book. Since the emphasis of the text is on the *use of high performance computers* rather than the *construction* of them, we will not go into details concerning hardware. Here we will emphasize the high end computers, because for such systems the differences between architectures are more easily visible. Smaller systems with similar characteristics also exist.

Since in modern high performance computers data movements between memory and arithmetic units is the most important factor for the efficiency of execution of a certain algorithm, our main classification scheme will distinguish between different architectures based on how these data transfers are organized.

First, there are now classical *pipelined vector computers*, with a moderate number of processors. These processors have vector instructions: only one machine instruction need be issued for the operations performed on one or two vectors of data. Often the data, on which vector instructions are performed are stored temporarily in vector registers. The primary memory is shared between the processors.

Second, there are multi-processor systems with comparatively simpler processors, connected in network, with *distributed memory* . The number of processors may range from, say, 20 to over thousands. Communication can take place in two different ways: In the *message passing model*, each processor can only access directly its own local memory, and when data are needed from another processor's memory, that processor must send a message with these data over the network. In the *data parallel programming model* there is one global address space that can be accessed by all processors. When a processor references an address outside its own memory, the run time system generates automatically the

communication need to transfer the data. The access time for non-local data is typically considerably longer than for local data.

The third class of multi-computer architectures is similar to the second in that the processors may be many and relatively simple, but here we have *shared memory*. The processors have one global address space, and they are are connected to the physical memory via some kind of switch. In principle, access times are the same for all processors and for the whole memory.

Technically speaking, classical supercomputers with vector instructions also belong to the latter category. Here we choose to treat them separately, since in most cases their individual processors are much more powerful than is common in the third class.

2.2 Flynn's Taxonomy

There exist quite a few classifications of computer architectures. In Flynn's taxonomy, the architectures are classified by the way processors execute their instructions on the data into the following four categories [14, 15]:

1. **SISD: Single Instruction stream Single Data stream**
   This model is nothing but the standard von Neuman organization described in Chapter 1. Instructions are fetched by the CPU one at a time, typically along with two data operands. After execution the resulting operand is stored to memory.

2. **SIMD: Single Instruction stream Multiple Data stream**
   An SIMD architecture has a single control processor that dispatches a single stream of instructions. The same instruction is executed simultaneously on many different streams of data. It does not distinguish between the different ways in the 'simultaneity' is achieved. Thus, this classification includes the pipelined vector as well as the array processors such as the Connection Machine and MasPar. An illustration of this model is in Figure 2.1 The SIMD computer offer free synchronization after each instruction execution and it is better for parallel programs that require frequent synchronization.

3. **MIMD: Multiple Instruction stream Multiple Data stream**
   The MIMD machines simultaneously execute different instructions on different data streams. The majority of multiprocessor configurations are included in this class [24]. In a MIMD computer, each processor has its own control unit and it is possible to use general purpose microprocessors in MIMD computers as processing units. MIMD computers offer a much higher degree of flexibility than SIMD computers since in MIMD computers, the processors can execute different codes, or if the same codes are executed, different branches can be chosen. Many unstructured applications are better suited to MIMD computer.

We now give a brief description of each of these architectures.

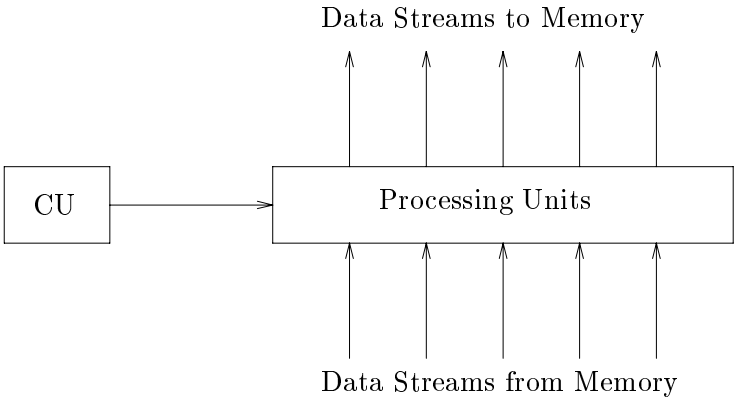


Figure 2.1: SIMD organization.

2.3 Pipelined Vector Computers

To take advantage of inherent parallelism in algorithms, the following forms of parallelism, among others, have been implemented in computer architectures.

1. Multiple functional units. This corresponds to duplicating the main functional units in the CPU, e.g., multiple adders and multipliers, and multiple I/O processors.

2. Pipelining and vector processing. When a large number of operations of the same type, e.g., additions, must be performed on a stream of data (a vector), there is no reason why one must until one operation is completed to perform the next one. The idea of pipelining is similar to that of an assembly line.

3. Multiple Vector pipelines.

These forms are often combined in a modern high performance computer. For example, multiple scalar units may be found along with several vector pipelines.

The term *vector computer* usually refers to a computer which includes vector operations in the instruction set. These operations are typically implemented with several 'vector' pipelines. In most cases the vector pipelines take their operands from vector registers, but there are architectures in which the operands are fetched directly from memory and returned to memory. Examples of vector computers include the family of Cray computers, CDC Cyber 205, IBM 3090, and NEC SX computers.

2.3.1 Pipelining

In this section we will discuss the concept of pipelining, and how it is used in vector instructions. We first illustrate **scalar computations** by taking the simplest possible example, the addition of two real numbers,

```
s1=s2+s3
```

The machine operations needed to execute this statement are (we use an informal assembler type notation. The number of cycles given should be considered only as an example).

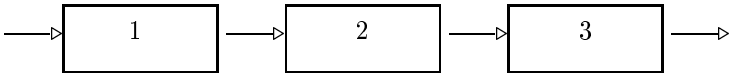
```
                          Number of cycles
                          ----------------

load  s2 --> R1                 1  % R1, R2, R3 are a registers
load  s3 --> R2                 1
add   R1 + R2 --> R3            6
store R3 --> s1                 1
```

In **scalar mode**, the execution of the following program

```
do i=1,1000
   s1(i)=s2(i)+s3(i)
enddo
```

takes 9000 cycles plus the overhead for the loop.

To introduce the concept of pipelining, consider an assembly line for making cars, and assume, for simplicity, that the line has only three stages, each of which takes equally long (one time unit).



The normal operation of such an assembly line is to input enough material for one car into the procedure every time unit, so that the workers are active all the time and produce one car every time unit.

When performing arithmetic operations, it is often the case that several stages must be performed before completion. The **functional unit** for the hardware that performs a specific arithmetic or logical operation in the computer. If the same operation must be applied to a stream of data there is no reason why one should wait for all the stages to be completed on one pair of operands before starting the next one. For example, assume that the floating point addition such as

$$1.234 \cdot 10^0 + 4.567 \cdot 10^{-2} = (1.234 + 0.04567) \cdot 10^0 = 1.27967 \cdot 10^0 \doteq 1.280 \cdot 10^0$$

consists of the following three suboperations

- Adjust exponents and fractions
- Add fractions
- Normalize fraction

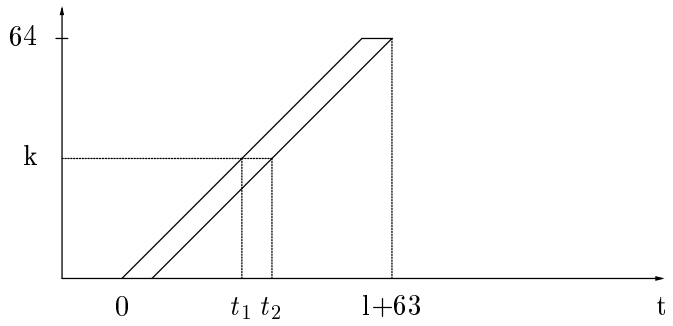
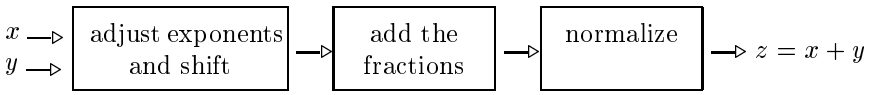


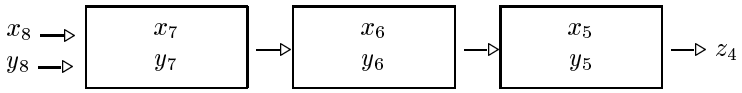
Figure 2.2: Timing diagram for a pipelined operation with $n = 64$. The pipe length is assumed to be l. The k'th operand pair enters the pipeline at clock cycle t_1 and leaves it at time t_2.

Then the operations can be divided into stages:



Assume that each stage takes one clock period. When the sum of two vectors $z := x + y$ (i.e., $z_i := x_i + y_i,\ i = 1, 2, \ldots, n$) is computed in a computer with pipelined floating point arithmetic, then the addition unit is operated like a car assembly line, **pipelining**. Then with a pair of input operands every clock period (after an initial startup time), an output is produced every clock period.

At a certain point in the computation, the operands have progressed through the pipeline as illustrated below.



After 3 cycles the first result emerges from the pipeline, and then one result is produced every clock cycle. A timing diagram for a pipelined operation is given in Figure 2.2

A **vector** is an ordered list of scalars. When a vector is used in a loop, this is often done with a constant distance, **stride**, between the elements referenced. In the code below, the vector is referenced with stride 1 on the first loop, and stride 3 in the second.

```
do i=1,1000
   a(i)=....
enddo
do i=1,1000,3
   b(i)=....
enddo
```

The computation in **vector mode** of the sum of the two vectors takes $l + n - 1$ clock periods, where l is the length (in cycles) of the pipeline and n is the vector length.

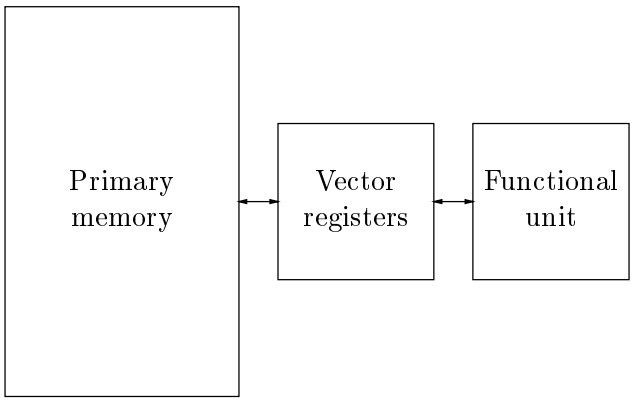


Figure 2.3: Vector registers are intermediate, fast storage units between primary memory and functional units.

The corresponding computation in **scalar mode** (without pipelining) would take ln clock periods. Thus, the speed-up is

$$s_n = \frac{nl}{l+n-1} \ .$$

For large values of n the speed-up becomes close to the length of the pipeline l.

2.3.2 Vector Register and Instructions

Pipelined functional units can produce one result every clock period, but they also need operands at the same rate. It is very expensive to construct memories that can deliver operands at this rate, and therefore most modern computers with pipelined arithmetic have **vector registers**, which can be considered as intermediate storage between the functional units and the primary memory.

In the Cray C90 processor, there are 8 vector registers, each with 64 elements. For our examples with vector registers we will assume that they have 64 elements. The vector registers are used in **vector instructions**. As an example, consider the following vector addition:

```
a(1:64)=b(1:64)+c(1:64)
```

We here use the array section syntax of Fortran 90, see Section 3.1.2. On computers with vector instructions the pseudo assembly code for would be

```
vload  b --> V1       % Vector load to the vector register V1
vload  c --> V2       % Vector load to the vector register V2
vadd   V1 + V2 --> V3 % Vector addition
vstore V3 --> a       % Vector store from V3 to memory
```

Thus, there are only four machine instructions. A more detailed assembly version of the same code can be

```
vload  b(1), 64, 1 --> V1        % Vector load to V1
vload  c(1), 64, 1 --> V2        % Vector load to V2
vadd   V1 + V2 --> V3            % Vector addition
vstore a(1), 64, 1 <-- V3        % Vector store from V3
```

where b(1) is the **start address** in memory of the vector b, 64 is the **vector length** (we assume that the vector registers have 64 elements), and 1 denotes the *stride*. Typically, a vector instruction reserves the output vector register for as long as the operation takes, and the input vector registers until the last element has been delivered to the vector functional unit. We say that a computation **vectorizes** if it can be performed with vector instructions. A compiler that can take a program written in a high level language and produce code with vector instructions is called a **vectorizing compiler** .

A vector operation cannot be stopped once it has started but as many operations will be performed as the vector length indicates. Therefore, loops with conditional statements cannot be vectorized (we will see later that there are methods that circumvent this problem).

A more serious difficulty is **recursion** :

```
do i=1,n
   x(i)=y(i)+x(i-1)
enddo
```

The same vector register can not be used both for input and output to the floating point functional unit, and therefore recursion can not be vectorized.

If the vector length in a vector operation is larger than the length of the vector registers, then the loop must be divided up in subloops. For example, in

```
x(1:n)=y(1:n)*z(1:n)
```

where $n > 64$, the compiler generates machine code, where the vector instructions have vector length equal to the length of the vector registers, i.e., 64 in our examples. Thus, the above code is replaced by the following:

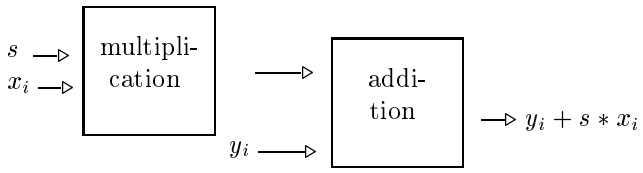
```
rem=mod(n,64)            % Remainder when n is divided by 64.
x(1:rem)=y(1:rem)+z(1:rem)
do j=rem+1,n,64
   x(j:j+63)=y(j:j+63)+z(j:j+63)
enddo
```

This technique is called **strip-mining** .

2.3.3 Chaining

Chaining is used in vector computers to further reduce the delay between consecutive pipelines. The idea is to take the result of one pipeline and direct it into a second pipeline

without waiting for all operations with the first pipeline to complete. In effect, this is nothing but the pipelining of two and possibly more pipelines which result in a single pipeline. By chaining the multiplication and addition units,



the computation of the so-called **Saxpy** operation,

```
y(1:64)=y(1:64)+s*x(1:64)
```

where `s` is a scalar, can be performed so that one result is produced every clock period. The result of the multiplication pipeline can be fed to the the addition pipeline as input, along with the input vector $y(1:n)$. Similarly, the multiplication and addition operations in the assignment

```
y(1:64)=y(1:64)+z(1:64)*x(1:64)
```

can be chained. On the vector instruction level we have (assuming that appropriate vector loads have been performed)

```
vmul  V1*V2 --> V3
vadd  V3+V4 --> V5
```

Chaining means that immediately after the vector multiplication has started, the addition is issued. However, it can not start until the first result has appeared from the multiplication pipeline. At the same time as the first result reaches V3 it also goes into the addition pipeline, and the addition can start. During this chained operation the multiplication and addition functional units work in *parallel* in a carefully synchronized manner.

In Figure 2.4 we give a timing diagram for two chained vector operations. For simplicity we assume that both operations have equal start up and unit times.

2.3.4 Performance Modeling of Vector Computers

A number of definitions concerning the performance of vector machines have been given by Hockney [25]. Let l denote the startup and unit time for a certain vector operation, i.e. the time to set up the vector instruction plus the time for the first pair of operands to pass through the pipeline. Then, the time to perform that operation on vectors with length n is

$$t = (l + n - 1)t_c,$$

where t_c is the cycle time. The rate of producing n results is

$$r_n = \frac{n}{(l + n - 1)t_c}.$$

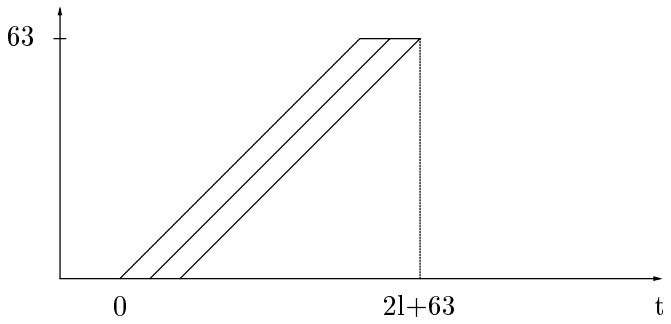


Figure 2.4: Timing diagram for two chained vector operations.

The **maximum** (or **asymptotic**) **rate** is obtained by letting n tend to infinity :

$$r_\infty = \frac{1}{t_c}.$$

Another interesting parameter is the **half performance length** $n_{1/2}$ which is the vector length required to achieve half the maximum performance. This can be determined from the equation

$$\frac{n}{(l+n-1)t_c} = \frac{r_\infty}{2} = \frac{1}{2t_c},$$

which gives

$$n_{1/2} = l - 1.$$

It is important to have a short start up and unit time l, since this determines the performance for relatively short vectors.

On a computer where two vector operations can be chained, after $2l$ cycles the *result of two floating point operations is output every cycle*. Therefore, on a computer where floating point addition and multiplication can be chained, the peak performance is

$$r_\infty = \frac{2}{t_c},$$

where t_c is the cycle time.

Example The CRAY X-MP had clock period $t_c = 8.5$ ns. This gives an asymptotic rate for, e.g., vector (componentwise) multiplication of $r_\infty = 1/8.5 \cdot 10^{-9} \approx 117$ Mflops (1 Mflop $= 10^6$ floating point operations).

The startup time l for multiplication is $l = 9$ clock periods. Therefore, the half performance length is $n_{1/2} = 9$. This indicates that the CRAY X-MP is very fast for short vectors also.

For the chained SAXPY operation $y := y + \alpha * x$, the asymptotic rate is $r_\infty \approx 234$ Mflops, since here the result of two arithmetic operations is output every clock period.

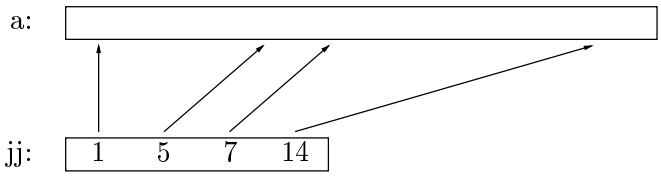


Figure 2.5: The vector `jj` with addresses to elements in `a`.

It should be emphasized that the values of these parameters are only *theoretical*. In practice, one has to take into account the time for memory accesses. The measured values are $r_\infty = 70,\ n_{1/2} = 53$ for vector multiplication, and $r_\infty = 148,\ n_{1/2} = 60$ for the SAXPY operation (from [24]).

2.3.5 Indirect Addressing

In sparse matrices, the majority of the elements are equal to zero. It is common in applications that such matrices of the order larger than 10^5 arise with the number of nonzero elements less than 5%. For such sparse matrices the usual matrix storage scheme should not be used, since the whole primary memory and much secondary memory would be wasted for storing zeros. Instead, only the nonzero elements can be stored, together with information about their location in the matrix.

In such applications the following type of code appears quite often:

```
do i=1,64
   a(jj(i))=b(jj(i))+s*c(jj(i))
enddo
```

`jj` is a vector of indices to elements in the vectors `a`, `b`, and `c`:

This is called **indirect addressing**. Many computers have vector instructions for indirect addressing operations:

```
vload jj    --> V0 % Load the index vector
vload c(V0) --> V1 % GATHER: load those elements of C, whose
                   % indices are in V0
vload b(V0) --> V2
vmult s*V1  --> V3
vadd  V3+V2 --> V4
vstore V4   --> a(V0)   % SCATTER
```

2.3.6 Conditional Statements

Since vector instructions cannot be interrupted, special arrangements need to be made in order to vectorize conditional statements. The **vector mask (VM) register** is a register with a number of positions that is the same as the length of the vector registers, each one bit wide. For example, the statement (the `where` statement in Fortran 90 is discussed in Section 3.1.4)

```
where (a(1:n) > 0) x(1:n)=y(1:n)*a(1:n)
```

can be vectorized using the VM registers as

```
vload  a --> V0
set VM to 1 where V0>0  % Otherwise 0
vload  y --> V1
vmul   V0*V1 --> V2     % Execute the multiplication for
                        % for all elements
vload  x --> V3
Generate V4 from V2 where VM=1 and from V3 where VM=0
vstore V4 --> x
```

where all instructions are vector operations. Note that, if only a few of the conditions are true, then many arithmetic operations are wasted with this construction and it may be much faster to execute the loop using scalar instructions. However the compiler cannot make a decision in advance. On some computers arbitrary vector operations can be controlled by the VM register.

If the computer implements the IEEE floating point standard, then the VM register can also be used for vectorizing the codes such as

```
where (a(1:n) > 0) x(1:n)=y(1:n)/a(1:n)
```

When implemented as in the above example, there may be divisions by zero, which will give infinity as a result. These can simply be masked away as above. If, however, the computer does not conform to the IEEE standard, then division by zero may interrupt the execution of the program. Therefore, on such computers, this example should be performed by scalar code.

2.3.7 Interleaved Memory

Typically, the **memory cycle time**, which is the time it takes for one word to be transferred from memory to a register, is larger than one cycle. For example, on the Cray X-MP the memory cycle time is four cycles. With such a memory speed, the vector load and store operations cannot deliver operands to and from vector registers at transfer speed that matches the speed of the vector floating operations.

As memory access patterns are important on high performance computers, we specify here how matrices are stored in primary memory. In Fortran, which is one of the most commonly used languages in scientific computations, matrices are stored in column major order. For example, a 3×3 matrix A is stored in the order

$$a_{11} \rightarrow a_{21} \rightarrow a_{31} \rightarrow a_{12} \rightarrow a_{22} \rightarrow a_{32} \rightarrow a_{13} \rightarrow a_{23} \rightarrow a_{33}$$

There are programming languages such as C and Pascal where matrices are stored in row major order. In the following, we assume the Fortran storage convention.

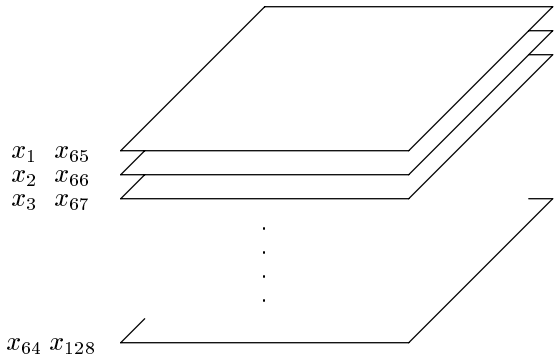


Figure 2.6: Storage of a vector with 128 elements in interleaved memory with 64 banks.

In the interleaved memory, organization, the memory is divided into separate units called banks in order to increase the performance of the memory. The banks can operate in parallel and each bank operate independently of each other. As an example, on the Cray C90 with primary memory of size 256 Mwords, the memory is interleaved with 256 banks. The consecutive elements of a vector are stored in consecutive banks, as shown in Figure 2.6.

Since the different banks can operate independently, it is possible to load consecutive elements of a vector from memory to vector registers (or the other way around), so that one word is delivered each clock cycle. On the other hand, if one loads non-consecutive elements of a vector, it may happen that when a word is requested from a memory bank, the bank has not finished processing the previous request. This is called a **memory bank conflict**. There is special hardware to resolve bank conflicts so that the second request must wait until the first is finished. Memory bank conflicts may occur whenever *non-unit stride* references to a vector are made. When matrices are stored in column major order, referencing a matrix row-wise results in referencing a vector with non-unit stride.

With interleaved memory `vload` (vector load) and `vstore` (vector store) operations can be chained. For example, the execution timing diagram of the assembly code

```
vload  b(1), 64, 1 --> V1        % Vector load to V1
vload  c(1), 64, 1 --> V2        % Vector load to V2
vadd   V1 + V2 --> V3            % Vector addition
vstore a(1), 64, 1 <-- V3
```

can be as shown in Figure 2.7.

If memory bank conflicts occur during the execution of a vector load or store operation, then the floating point operations are delayed. In order to perform the above code as a chained vector operations there must be two read channels and one write channel between the primary memory and the vector registers, provided that the memory can deliver operands at high enough speed.

A similar type of conflict occurs if two different processors access the same bank in a computer with multiple processors, and the conflict is resolved by special hardware. Note that interleaved memory with many banks is particularly useful in computers where

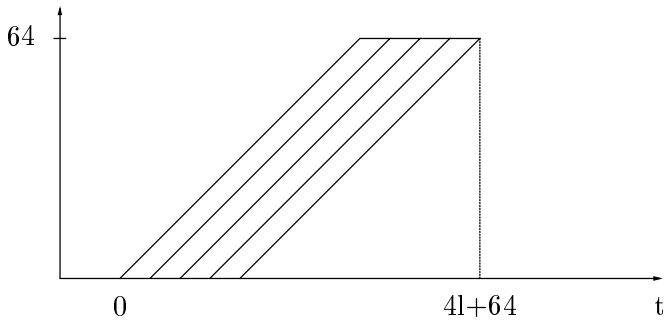


Figure 2.7: Timing diagram for four chained vector operations.

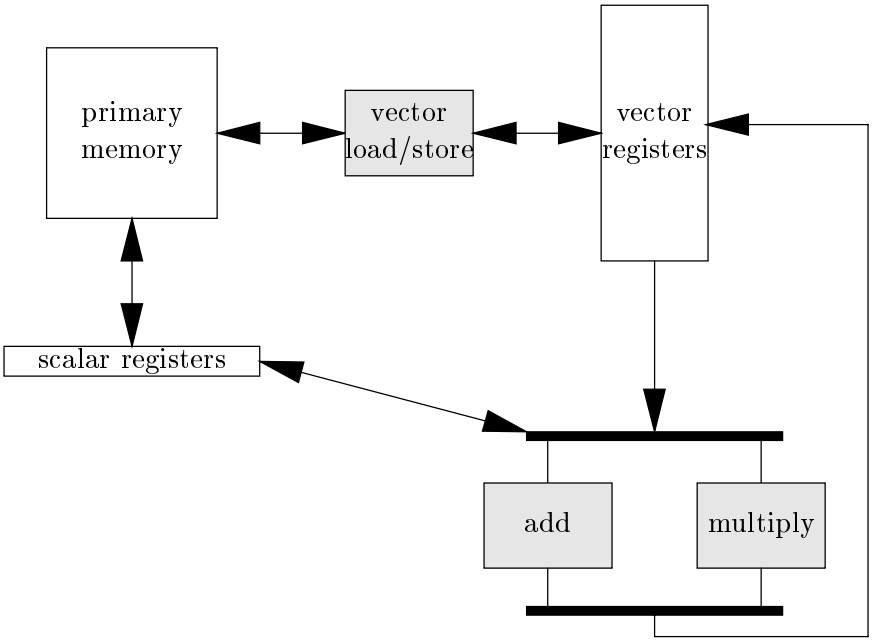


Figure 2.8: Vector register architecture. Pipelined units are shaded.

several processors share the same memory, since this reduces the risk of this type of memory conflict. Figure 2.8 shows the basic structure of a vector register architecture.

2.4 Memory Organization

Along with the progress on the control processing units, some advances in memory organization took place. The memory organizations on multiprocessor systems can be classified into shared memory and distributed memory. One of the pioneers in the early history of the digital computer stated

> *In my opinion this problem of making a large memory available at reasonably short notice is much more important than that of doing operations such as multiplication at high speed.* (Alan Turing, 1947)

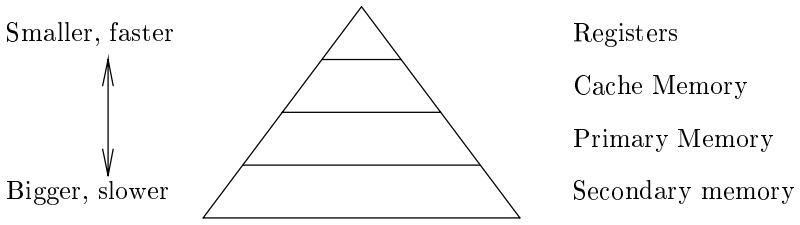


Figure 2.9: Memory hierarchy

This statement is equally true today: to balance the speed of the floating point functional units and the parallel features (multiple pipelines, multiple processors), large memories are needed, typically of the order 0.1 to 1 Gwords which is 10^9 words (64 bit). Since the fast memory is expensive, compromises have to be made between size and speed.

2.4.1 Memory Hierarchy

One of the most important concepts in high performance computers is **memory hierarchy**. Due to the cost of manufacturing very fast memory hardware, computer designers often compromise between memory speed and memory size. Many modern high performance computers, therefore, have a memory hierarchy as shown in Figure 2.9. Some computers have vector registers and no cache memory , while others have cache and no vector registers. There are computers that have both.

A cache is a fast memory between the processor and the primary memory, see Figure 2.10. The references are often made to the same memory location several times and also when a certain memory location has been accessed, then it is very likely that nearby locations will be accessed soon. Therefore, by storing a portion of the primary memory in the cache memory, which can deliver operands to and from the processor much faster than the primary memory, the overall speed can be improved.

The minimum unit of information that is handled by this two-level memory hierarchy is a *block*. A certain number of blocks from the primary memory are stored in the cache. A special memory, called the *tag memory*, where all locations can be read in parallel, is used to store the block addresses which are the leading bits of all the memory locations of the block.

Once a block of data is brought the to cache it remains there until it becomes necessary to bring in another block, which must overwrite it.

When a memory access is found in the cache, it is called a *cache hit*. If a memory access results in a *miss* , then the block containing that memory location is copied to the cache before the item referenced can be transported to or from the processor.

For more detailed descriptions of cache memories, see [22, 26].

It is desirable to *perform as many floating point operations as possible for every floating point variable that is transferred from primary memory to the registers or cache memory.*

2.4.2 Shared versus Distributed Memory

- *Shared memory models* : processors have very little local or 'private' memory; they

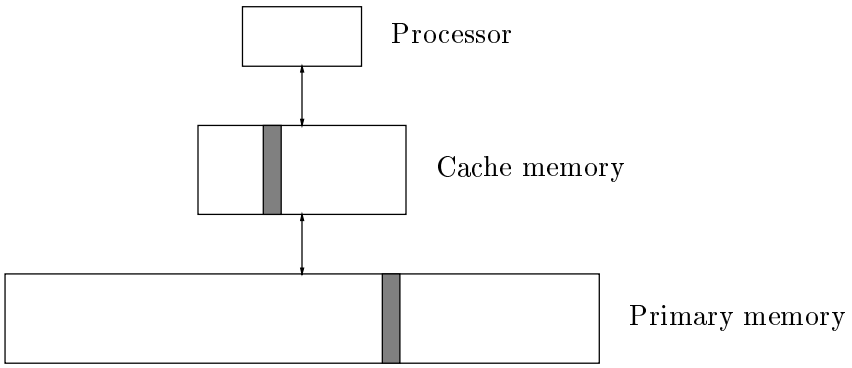


Figure 2.10: Two-level memory hierarchy. The shaded area represents a block from primary memory that has been copied to the cache memory.

exchange data and co-operate by accessing a global shared memory.

- *Distributed, local memory models* : there is no global memory that is shared by all processors. Instead each processor will have a sizable local memory and generally no access to other processor's local memory. Physical interconnections between certain processors allow for exchange of data and control information. Any synchronization results from a possible need for exchanging data with other processors. The processing elements have their own control units, but these units will communicate with other control units.

- *Distributed, shared memory models* : memory is physically distributed among the processors, but there is a global address space. When a processor makes a reference to a memory position that is physically within another processor, then the hardware automatically does the communication needed.

2.5 Shared Memory Multiprocessors

Multiprocessor architectures with shared memory are tightly coupled systems in which the processors are connected to a large global memory. All the processors have the same view of the address space. In addition, the usual assumption for shared memory models is that access to data does not depend on its location in memory. In a shared memory environment programming is greatly facilitated due to transparent data access; from the user's point of view data are stored in a large global memory readily accessible to any processor.

The primary memory may be centralized to have only one memory module or partitioned into several modules. The interconnection network is a potential bottleneck for these systems. Also, memory conflicts can lead to degraded performance because of the need of many processors to simultaneously access the same memory locations. The interaction between processors and processes are controlled by a common operating system.

The major limitation of the shared-memory system is the possibility of primary memory access conflicts and this tends to put an upper bound on the number of processors that can be effectively incorporated in the system. In addition, shared memory computers cannot easily take advantage of proximity of data in problems with local (data) dependences. There are two possible implementations of shared memory machines. The first uses a high speed bus to connect memories to processors and the second uses a switch.

2.5.1 Bus-based Shared Memory Multiprocess

Shared memory computers are more often implemented with buses than with switching networks. Busses are the backbone for communication between the different units of most computers. Physically, a bus is a collection of wires, made of either fiber or copper. These wires will simultaneously carry information consisting of data, control signals, and error correction bits. The speed of a bus, often measured in Megabytes per second and called the *bandwidth* of the bus, is determined by the number of lines in the bus and the clock rate. Often the limiting factor in machines based on bus architectures is the bus bandwidth rather than the CPU speed.

If the bandwidth of each of the uniprocessor buses is B_P and that of the global bus is B_G then, generally the limit due to communication bandwidth is set by the minimum between B_P and B_G/p.

The primary reason why bus-based multiprocessors are more common than switch-based ones is that the hardware involved in such implementations is much simpler. On the other hand, the difficulty with bus based machines is that the number of processors which can be connected to the memory will be small in general, unless one designs a network of buses as was done in the 1970's with Burroughs computers. The bus is typically timeshared, in that slices of time are allocated to the different clients such as processors and I/O processors that request its use. In a multiprocessor environment, the bus can easily be saturated due to large communication demand. Several remedies are possible. The first is to reduce traffic by adding local memories or caches attached to each processor. This remedy in turn causes some difficulties due to the coherence of data. If local memory contains some data that has just been brought from memory, and another processor modifies that data after it has been read by the processor, then we end up with two copies of the same data that have different values. A mechanism should be put in place to ensure that the most recent update of the data is being used. It may well be the case that the additional overhead incurred by the maintenance of memory coherence will offset the gain due to savings in memory traffic. The second remedy is to organize the architecture around an array of buses as opposed to a single shared bus.

Typically, the bus is capable of sending data packets from one processor to all other processors in the same amount of time as it would send them to just one processor. Each processor may have its own local memory which is large enough to hold the data and code of typical application programs run on the machine.

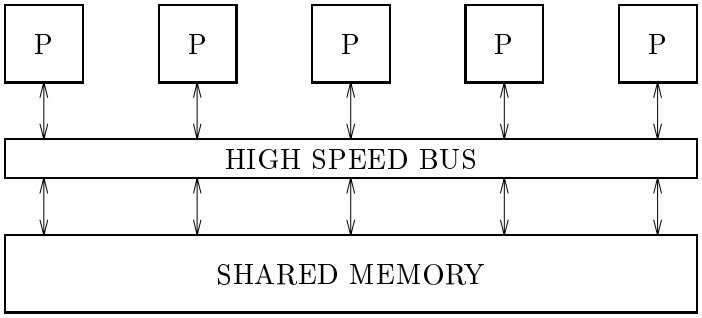


Figure 2.11: A bus-based shared memory computer.

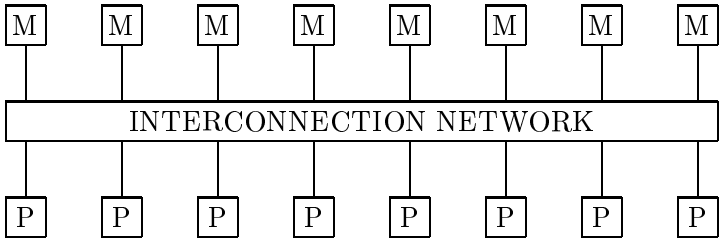


Figure 2.12: Switch-based Shared Memory Computer

2.5.2 Switch-based Shared Memory Multiprocessors

The diagram of a shared memory multiprocessor based on a switching network is shown in Figure 2.12. Variations on this scheme are numerous, but the essential features here are the switching network and the shared memory. Examples include IBM RP3 which uses an Omega network and the BBN butterfly computer. The switching network can be a cross-bar switch when the number of processors is small. When there is a large number of processors, the switch is usually a packet switching multistage network. Packets of data are driven across a small number of stages consisting of an array of elementary switches, e.g., 2×2 or 4×4 switches. The routing of the packet is determined by an address tag that is part of the packet.

Figure 2.13 shows Pease's indirect n-cube network which is topologically equivalent to the binary n-cube. Each of the three stages D_1, D_2, and D_3 corresponds to the position of the bit that determines the direction of the packet. These bits are considered from right (least significant) to left (most significant). A zero bit in the i-th position dictates that the packet take the up direction in the switch, upon reaching a switch in the i-th stage.

For example, a packet that must travel from node 110 to node 010 must first go up at the '0' exit of the bottom switch in D_1 to enter the third switch (from top) in D_2. Then it

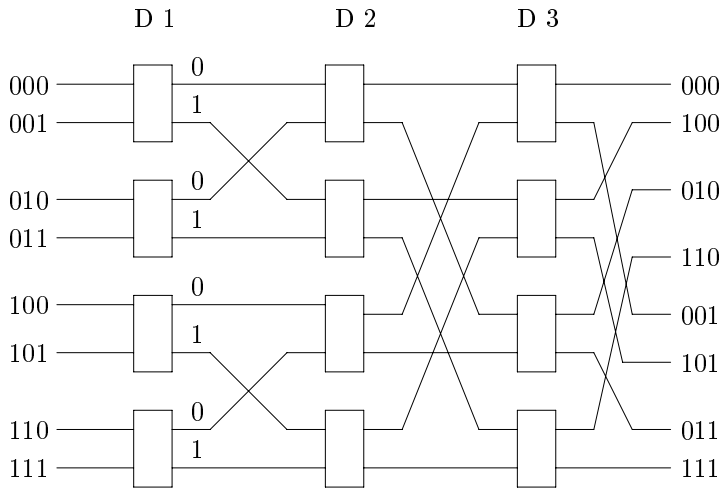


Figure 2.13: The Pease multistage network for 8 processors

is directed to the down position of this switch ('1' exit) towards the third switch in D_3. It then exits at the up position ('0' exit) of this switch to finally reach the destination 010. Omega networks and Pease networks are identical in concept. Other types of network exist but many of them are topologically equivalent.

A network of this type is able to simulate any interconnection topology, i.e., any permutation of the p inputs to p outputs can be realized, possibly by a small number of passes across the switch. There are also multistage networks that can realize any permutation in one pass but they are more complicated to set-up. The switching network becomes exceedingly complex as the number of processors and memories increases : the connection of N processors to N memories in general requires a total of $O(N \log_2 N)$ identical 2×2 switches.

2.5.3 Circuit Switching and Packet Switching

There has been two ways of exploiting multistage switching networks described in previous section. The first is circuit switching. In circuit switching the elementary switches are set-up by sending electronic signals across all of the switches. The circuit is set-up once in a similar way in which the telephone circuits are switched. Once the switch has been set-up, the communication between processors $P_1, \ldots, P_n$ is open to the memories

$$M_{\pi_1}, M_{\pi_2}, \ldots, M_{\pi_n} \ ,$$

where π represents the desired permutation. This communication will remain functional as long as it is needed. When a new permutation is desired, the switch will be reset. Setting up the switch can be costly but once it is set then communication can be quite fast. This approach is taken in the GF11 computer built by IBM.

In packet switching networks, a packet of data will be given an address token and the switching within the stages will be done based on the value of the token. This was illustrated in the previous examples.

2.5.4 Communication in Shared Memory Computers

In a shared memory model the p processors are linked to a global shared memory which they can ideally access with equal speed. Often each processor is provided with its local memory but this is used only as temporary storage and is not assumed to be large enough to hold the data of the problem to be solved, which is stored in the global memory. If the communication bandwidth of each processor is b_M, the time that it will take to move a packet of m words between any processor and the memory is of the form

$$t_m = \tau_M + \beta_M m,$$

whether the architecture is based on a bus or a switch. However, the above model is simplistic since it assumes that there are no bus contentions, memory conflicts, or hot-spots, i.e., contention at the level of the switches in packet-switched networks. Often the memory is divided into memory banks to allow efficient access of the memory by several processors. For a switch based machine, we can assume that as long as there is no contention to the same memory bank, several processors can read and write simultaneously. On bus-based architectures, it is usually the case that the bandwidth of the global memory does not exceed a multiple k^* of b_M, where $k^* \leq k$. This means that at most k^* different processors can access the memory with equal speed b_M. This is the best possible scenario where there is no memory bank conflict. Memory bank contention can ruin performance and one of the goals for a software developer is to arrange the data initially so as to minimize memory conflicts [36, 17].

2.6 Distributed Memory Multiprocessors

By *distributed memory* architectures we will refer to the distributed memory *message passing* architectures as well as to distributed memory SIMD computers. Distributed memory parallel computers are efficient for problems that can be partitioned into larger tasks that do not interact very frequently. A typical distributed memory system consists of a large number of identical processors which have their own memories and are interconnected in a certain topology. The processors are linked to a number of 'neighboring' processors which in turn are linked to other neighboring processors. Each message usually consists of a number of fixed-size packets, and the inter-processor communication follows a predetermined communication protocol, cf. Section 3.2. In 'Message Passing' models computations are data driven, i.e., a computation in a particular processor is performed only when the necessary operands become available and there is no global synchronization. Data communication is carried out through message passing.

2.6.1 Communication in Distributed Memory Computers

The following data exchange operations are common in many numerical algorithms.

1. Moving data from one processor to another. This represents the simple *one-to-one data transfer operation.*

2. Moving the same data packet from one processor to all others. We will refer to this as a *broadcast operation.*

3. Scattering data from one processor to all others and gathering data in one processor from all others. In the *scatter operation*, a node sends packets to every other processor. These packets, although different, are ideally of the same size. The *gather operation* is the dual operation: the node receives a packet of (ideally) equal size from each of the other nodes.

The difference between the broadcast (2) and the scatter (3) is that in the scatter operations a different data set is sent to each processor. The gather and scatter operations are very similar in nature. An algorithm for scatter can be derived from an algorithm for gathering simply reversing the data paths and vice-versa. For this reason we will only consider scatter operations.

In many present day parallel computers with distributed memory the communication hardware is such that the time for sending a message between two nodes is more or less the same for any pair of nodes in the processor network, cf. [28, p. 46]. In many cases it is the startup time for a communication that dominates [16, p. 91, Table 3.1]. Therefore it is reasonable to make the assumption that a data packet of size m can be moved from one node to any other node in time

$$t(m) = \tau + \beta m \tag{2.1}$$

The constants τ and β are dependent on the system, where τ is the communication start-up or latency and β depends on the bandwidth of the channel between the two nodes. Note that we assume that *a processor can only send to one other processor at the same time.*

In most cases, the start-up time τ is much larger than the time for a floating point operation, γ. Typically (1998) γ is of the order 10^{-7}–10^{-8}, while τ can be as large as 10^{-4}–10^{-5}. In Table 2.1 we give the parameters for two parallel computers.

	γ	τ	β
IBM SP	0.01	50	0.01
Cray T3E-900	0.01	16	0.0065

Table 2.1: Typical arithmetic and communication parameters (in μs) for two parallel computers. The time for an arithmetic operation is an average value over different operations and problem sizes.

When discussing the time required for broadcast operations, it is important to make a difference between the physical and the logical topology of a processor network. For instance, broadcasting a message to the processors in a *physical* ring, can be done by sending it from processor 0 to 1, then from 1 to 2, etc. This takes

$$p\,(\tau + \beta m),$$

where p is the number of processors. This type of broadcast is often used when a ring of processes is organized as a pipeline for computation and communication. We will refer to this as a *pipelined broadcast*. However, if a *logical* ring is embedded in a physical network topology, e.g. a hypercube, such that (2.1) is satisfied, then a broadcast can be executed in

$$\log_2 p(\tau + \beta m),$$

where the number of processors p is assumed to be a power of two. The following code segment performs a broadcast in $\log_2 p$ steps; the processors are assumed to be numbered from 0 to $p-1$, and at the beginning processor 0 holds the item `data` that is to be broadcast. In the code we use message passing terminology, see Section 3.2; in reality broadcast operation are often given as communication primitives. The variable `myid` is the number of the processor.

```
k=2log(p)                        ! logarithm base 2
do i=0,k-1
   if myid < 2**i then
      send(2**i + myid, data)
   else if myid < 2**(i+1)
      receive(data)
   endif
enddo
```

The communications performed in the first three steps are

i	communication
0	$0 \to 1$
1	$0 \to 2$
	$1 \to 3$
2	$0 \to 4$
	$1 \to 5$
	$2 \to 6$
	$3 \to 7$

Since this broadcast algorithm can be illustrated by a binary tree, we will refer to this as a *tree broadcast*. The timings of the two types of broadcasts seem to indicate that always a tree broadcast is to be preferred. However, in the subsequent chapters we will show that in many cases the pipelined broadcast combined with computations gives more efficient algorithms.

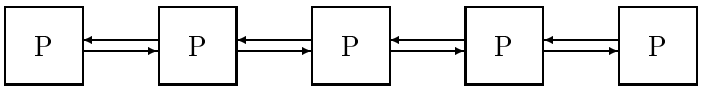


Figure 2.14: A Linear Array of Five Processors.

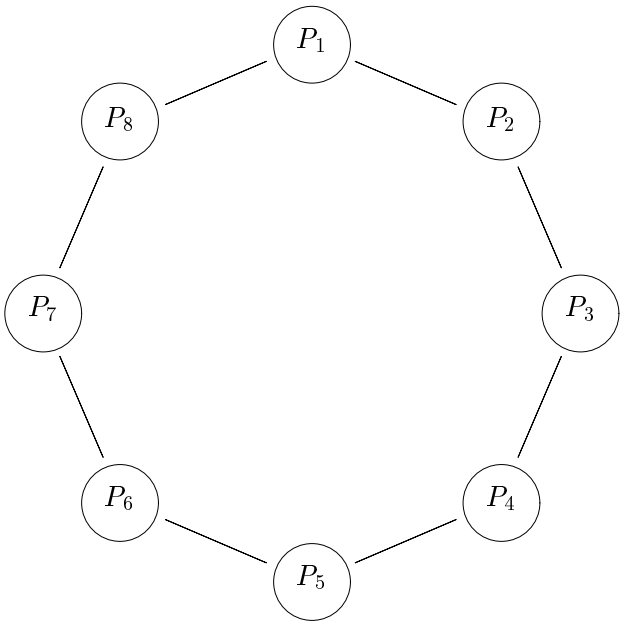


Figure 2.15: A Processor Ring Consisting of Six Processors.

2.6.2 One-, Two- and Three-Dimensional Arrays

In a linear array, processors are connected along a line as shown in Figure 2.15. When two processors at the left and right ends are connected, the multiprocessors make a ring. Each processor can simultaneously write to both neighbors or simultaneously read from one neighbor and write to the other. In a linear array of processors, the communication of two boundary processors may be taken care of separately since they have only one neighbor each unlike the other processors.

A square two-dimensional grid consists of an array of processors connected as shown in Figure 2.16. In this book we will only consider two-dimensional arrays. Many of the algorithms developed for two-dimensional array can be extended to three-dimensional arrays. Two-dimensional and three-dimensional arrays are popular among partial differential equations specialists because they offer a simple way of mapping regular finite difference grids into the multiprocessor grid.

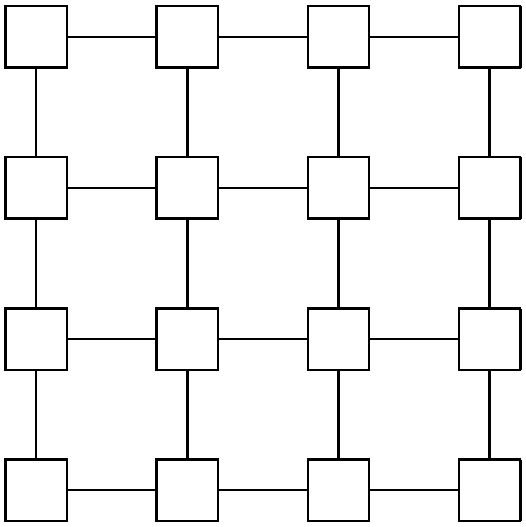


Figure 2.16: A 4×4 two-dimensional array of processors

Often, the processors on each side of the grid are connected to those on the opposite side forming the wrap around connections which yields more homogeneous complexity results.

Chapter 3

Programming Models and Environments

This chapter gives an overview of different software-related concepts in vectorization and parallelization. Thus we introduce programming models for each of the most important architectures that exist today. We would like to have conceptual models of the existing machines that can be used for designing the algorithms and for analyzing them. The following programming models/environments will be treated:

- *Fortran 90.* The latest Fortran standards have been designed to express parallelism and vectorization.
- *Message passing and MPI.* This is the most common programming model for distributed memory parallel computers.
- *Shared memory parallelism and OpenMP.* Sequential codes can be parallelized for shared memory systems by compiler directives.
- *Data parallel programming.* A powerful programming model based on the fact that in many cases the order of certain computations is irrelevant.

3.1 Fortran 90

Fortran, created in the late 1950s, is still one of the most widely used programming languages for solving problems in science and engineering. A Fortran standard was adopted in 1991 and it is called Fortran 90 (the previous one was Fortran 77). It has several constructs that are aimed at making vectorization and parallelization easier. Here we briefly describe a few features in Fortran 90, which are important for vector and parallel computers. For a more comprehensive description of Fortran 90[1], see e.g. [4], [32].

[1] Another Fortran standard was adopted in 1995.

3.1.1 Vectors and Matrices

Many programming languages, including Fortran 77, require the programmer to specify in to great detail the order in which computations are to be performed. Consider the following Fortran 77 code segment

```
      do 10 i=1,n
         a(i)=b(i)+c(i)
10    continue
```

The semantics of the language prescribes that first `a(1)` is computed, then `a(2)`, etc. Now, since each iteration of the loop is completely independent of the others, the computation in itself need not be performed in any particular order. However, Fortran 77 forces the programmer to *overspecify* the order of computation. The same is true of many other languages.

In Fortran 90 overspecification of computation order is avoided be defing arrays (vectors and matrices) to be data objects in themselves, and they can be referenced as such, not just as a collection of subscripted scalars.

Let `a`, `b` and `c` be declared

```
real, dimension(1:m,1:n)       :: a, b, c
```

Arithmetic operations for matrices can be written

```
a=b*c
c=a-b
```

Here, the multiplication is *element-wise*, which is different from matrix-matrix multiplication. It is up to the compiler writer to decide the order of computation of the elements of the matrices, which, for instance, can be made to depend on the architecture of the computer. Note however, that the first statement `a=b*c` must be completed before the second can start.

3.1.2 Array Sections

Array sections can be referenced using a notation analogous to that in the `do`-loop:

```
i:j:k
```

where `i` is the start index , `j` the final index and `k` is the stride. For example,

```
x(1:20:2)=y(1:10)
```

which is a Fortran 90 statement, can be written in Fortran 77 as

```
do i=1,10
   i1=(i-1)*2+1
   x(i1)=y(i)
enddo
```

There is one important difference here, however: *using the array section, the order in which the operations to the elements are performed is not prescribed.* The variant

```
i:j
```

means that stride 1 is assumed,

```
 :j
```

assumes the lower limit in the declaration, and

```
 :
```

means that both the lower and upper limits in the declaration are assumed. Thus, if `a` is declared

```
real a(100)
```

then the following references are equivalent

```
a(1:100:1), a(1:100), a(:100), a(1:), a(:), a
```

and they all refer to the whole vector.

Let the matrix `x` be declared

```
x(1:100,1:50)
```

Then

```
x(1:50,1:10)
```

is a reference to the upper left submatrix of dimensions 50×10, and

```
x(:,25:30:2)
```

is a reference to the column vectors `x(1:100,25)`, `x(1:100,27)` and `x(1:100,29)`.

If a scalar is used in a assignment statement together with arrays, it is considered as an array of an appropriate dimension. The assignment

```
a(:)=3.14
```

gives each element of `a` the value 3.14. A further example is as follows:

```
real a(100), b(-1:98), x(100,50,25), y(100,100,10,70), p,q
....
a=1.0
b(:10)=p+q
x(:,n,1)=a+b
x(m:n,1:10,1:20)=y(1,m:n,1:10,41:60)
```

Note that all arrays used in an assignment must have conforming dimensions.

The following rule is important in understanding the difference between array sections used in assignments and `do` command, and in understanding how array sections are executed using vector instructions. In the assignment

```
array section=expression
```

the whole expression in the right hand side is computed before the assignment takes place. Thus, the assignment

```
a(2:n)=a(1:n-1)+a(3:n+1)
```

is **not** equivalent to

```
do i=2,n
   a(i)=a(i-1)+a(i+1)
enddo
```

but to

```
do i=2,n
   temp(i)=a(i-1)+a(i+1)
enddo
do i=2,n
   a(i)=temp(i)
enddo
```

in the sense that they give the same results.

3.1.3 Array functions

All the intrinsic functions can be used for arrays. Let `a` be declared as above. The statement

```
b=sin(a)
```

produces a matrix `b`, the elements of which are

```
b(i,j)=sin(a(i,j))
```

In addition to the library functions from Fortran 77, there are functions for performing common vector and matrix operations such as

`s=dot_product(x,y)` Scalar product of the two vectors `x` and `y`.

`s=sum(x(1:n))` Summation of the components in `x`.

`c=matmul(a,b)` Matrix multiplication, see below.

In the following code we compute the matrix product $C = AB$, where all three matrices are square, in three different ways, using the above functions.

```
real, dimension(1:100,1:100)   :: a,b,c1,c2,c3
integer                        :: i,j

.....          ! Variables are given values

do i=1,100
   do j=1,100
      c1(i,j)=sum(a(i,:)*b(:,j))
   enddo
enddo

do i=1,100
   do j=1,100
      c2(i,j)=dot_product(a(i,:),b(:,j))
   enddo
enddo

c3=matmul(a,b)
```

Shift functions take arrays as input and give arrays, where the elements have been rearranged as output. We consider a vector example.

```
real, dimension (1:4)   :: x, y, z
x=(/1,2,3,4/)
y=eoshift(x,shift=1)                  ! End-off shift
z=cshift(x,shift=-1)                  ! Circular shift
```

After executing this code the result is `y=(2,3,4,0)` and `z=(4,1,2,3)`, where shift = 1 and -1 and it denotes the left and right shifts, respectively. Shift functions applied to matrices (and arrays of higher dimension) are analogous, as seen in the following example.

```
real, dimension (1:3,1:3)   :: x, y
integer                     :: i

x=reshape((/1,2,3,4,5,6,7,8,9/),(/3,3/))

write(*,*)'x before shift'
do i=1,3
   write(*,*)x(i,1:3)
enddo

! Circular row shift to the left
```

```
y=cshift(x,shift=(/1,2,3/),dim=2)

write(*,*)'after shift'
do i=1,3
   write(*,*)y(i,1:3)
enddo
end
```

The `reshape` function takes a vector as input and creates a 3 by 3 matrix `X` and `cshift` shifts along the second dimension, i.e. row-wise. The first row is shifted one step, the secoond two steps, and the third three steps. The output of the code is

```
 x before shift
   1.0000000   4.0000000   7.0000000
   2.0000000   5.0000000   8.0000000
   3.0000000   6.0000000   9.0000000
 after shift
   4.0000000   7.0000000   1.0000000
   8.0000000   2.0000000   5.0000000
   3.0000000   6.0000000   9.0000000
```

The function `spread` can be used to create a matrix from a vector by replicating the vector elements by rows or columns. The code

```
real, dimension (1:4,1:3)  :: a
real, dimension (1:3,1:4)  :: b
real, dimension (1:4)      :: x
integer                    :: i

x=(/1,2,3,4/)
b=spread(x,1,3)        ! Make 3 copies of x along the first dimension,
                       ! i.e., by columns.
write(*,*)'b'
do i=1,3
   write(*,*)b(i,1:4)
enddo

a=spread(x,2,3)        ! Make 3 copies of x along the second dimension,
                       ! i.e., by rows.
write(*,*)'a'
do i=1,4
   write(*,*)a(i,1:3)
```

```
      enddo
```

gives as result

```
 b
   1.0000000   2.0000000   3.0000000   4.0000000
   1.0000000   2.0000000   3.0000000   4.0000000
   1.0000000   2.0000000   3.0000000   4.0000000
 a
   1.0000000   1.0000000   1.0000000
   2.0000000   2.0000000   2.0000000
   3.0000000   3.0000000   3.0000000
   4.0000000   4.0000000   4.0000000
```

3.1.4 Vector Mask Operations

The statement

```
     where (a(1:n) >  b(1:n)) a(1:n)=x
```

gives the same result as

```
      do i=1,n
         if (a(i) .gt. b(i)) a(i)=x
      enddo
```

Similarly

```
      where (a(1:n) >  b(1:n))
         a(1:n)=x
      elsewhere
         a(1:n)=b(1:n)
      endwhere
```

gives the same result as

```
      do i=1,n
         if (a(i) .gt. b(i)) then
            a(i)=x
         else
            a(i)=b(i)
         endif
      enddo
```

These code sections can be implemented on vector machines using vector mask operations. The intrinsic functions can also be used together with mask operations. For example, the statement

```
s=sum(x(1:n), where x > 0)
```

sums the positive components of the vector `x`.

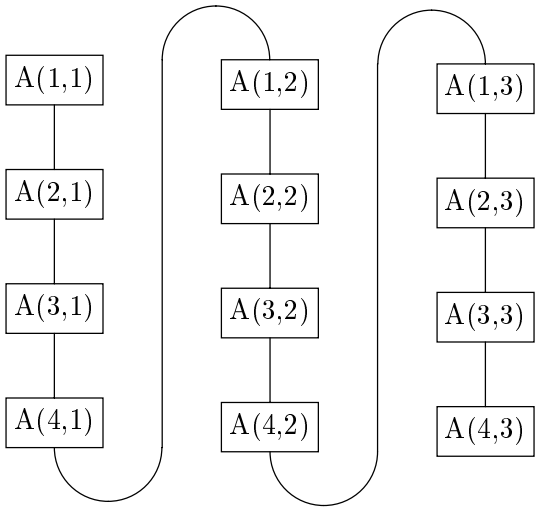
3.1.5 Vectorization of Fortran Codes

3.1.5.1 Storage of Matrices

In Fortran matrices are stored in column major order. E.g., a matrix `A` declared as

```
real A(1:4,1:3)
```

is stored



If we reference the matrix column-wise,

```
do j=1,3
   do i=1,4
      a(i,j)=...
   enddo
enddo
```

then we have stride 1. If we reference row-wise

```
do i=1,4
   do j=1,3
      a(i,j)=...
   enddo
enddo
```

then we have stride 4. This storage mode has consequences for high performance computers. For instance, assume that we are using an interleaved memory computer with 16 banks, and that we access a 16×16 matrix by rows. Then memory bank conflicts will occur. In general, the risk for memory bank conflicts is lower, if the matrix is accessed by columns.

Similarly, if the computer has a cache memory, then the blocks in the cache will contain "vertical" slices of a large matrix, and row-wise access will lead to many cache misses.

We conclude that *in order to execute efficiently on high performance computers, matrix algorithms programmed in Fortran should access the matrices column-wise.*

3.1.5.2 Vectorization of Loops

In general, Fortran code where the assignments can be expressed with array sections can be executed using vector instructions. However, not all algorithms are or can be expressed conveniently using array sections. The task of a **vectorizing compiler** is to analyze do loops, and generate vector instructions where this is possible.

3.1.5.2.1 Vector Reference Earlier we saw that a memory reference for a vector, i.e. a vector load or store, has a start address, a length (the number of words that need to be transferred), and a constant stride. In principle we have

```
vload x(1), VL, stride --> Vreg
```

`VL` elements from the vector `x` are loaded to the vector register `Vreg`, starting with element `x(1)`. The stride is `stride`.

The following definitions are taken from [29].

1. An integer variable, which has a constant increment in a loop, is called a `CII` (Constant Increment Integer).

2. A **vector reference** is a reference inside a loop where all indices are of the form

$$[\pm \texttt{invariant expression} *] \texttt{ CII } [\pm \texttt{invariant expression}]$$

It is easy to see that with these definitions all vector references have a start address, a vector length, and a constant stride.

In the example

```
real x(500), a(500,250), b(1000)
do k=1,n
   i=3*k+n
   j=m*k+6
   a(i,j)=x(k)+b(j-4)
enddo
```

all the references to arrays are vector references:

array	start	length	stride
a	a(n+3,m+6)	n	m*500+3
x	x(1)	n	1
b	b(m+2)	n	m

Loops where all array references are vector references can be executed using vector instructions.

3.1.5.2.2 Recursion Since the semantics of Fortran prescribe sequential execution, data dependence between two Fortran statements of the type

```
x=a*b
y=x*z
```

implies that the statements must be executed in this order. When a loop is executed, then it assumed that the iterations are performed in the order specified in the `do` statement. Therefore, a loop can be vectorized if no data are used that have been modified in a previous iteration.

It is obvious that the iterations in the code

```
do i=1,n
   a(i)=b(i)+1.0
enddo
```

are completely independent and can be vectorized (and also parallelized). Similarly, the following loop can be vectorized

```
do i=1,n
   a(i)=a(i+1)+1.0
enddo
```

since the elements on the right hand side in the assignment are *unmodified during previous iterations of the loop.* However, in the loop

```
do i=1,n
   a(i)=a(i-1)+1.0
enddo
```

elements that have previously been modified are on the right hand side. This is called **recursion** and cannot be vectorized.

An example of a very important application, where recursion occurs, is the solution of a bidiagonal linear system of equations

$$\begin{pmatrix} a_1 & & & & \\ b_2 & a_2 & & & \\ & b_3 & a_3 & & \\ & & \ddots & \ddots & \\ & & & b_n & a_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{pmatrix}$$

This can be solved by the code

```
x(1)=d(1)/a(1)
do i=2,n
   x(i)=(d(i)-b(i)*x(i-1))/a(i)
enddo
```

Such a recursion can only be executed in scalar mode.

In order for the compiler to generate vector instructions, it must be clear at compile time that the code can be vectorized. Consider

```
do i=1,n
   x(i)=c(i)*5.0
   y(i)=a(i)*x(i+k)
enddo
```

Here it is in general impossible for the compiler to determine if `k` will be negative or positive (unless `k` is explicitly assigned a constant value in the program, and this is the only assignment where it occurs). Similarly, the compiler will have difficulties with the indirect addressing in the first statement of the following code.

```
do i=1,n
   x(i)=c(i)+y(iy(i))
   y(i)=a(i)+1.0
enddo
```

It is possible that for some previous `i`, `y(iy(i))` has been modified (e.g., if `iy(2)=1`).

In such cases the compiler cannot decide if recursion will take place or not. But if the programmer knows that no recursion will occur, then he/she should give the compiler directives to vectorize (and he/she becomes responsible for errors, not the compiler).

3.1.5.2.3 Indirect Addressing Indirect addressing means that a vector is referenced via a vector of indices. In Fortran 77 we write

```
do i=1,64
   x(ix(i))=y(iy(i))+z(iz(i))
enddo
```

where `ia`, `ib`, and `ic` are integer arrays holding the indices of the elements in the arrays that we use in the assignment statement. The corresponding Fortran 90 code is

```
x(ix(1:64))=y(iy(1:64))+z(iz(1:64))
```

Loading data this way (`y` and `z`) is called **gather**, and storing (`x`) is called **scatter**, cf. Section 2.3.5. Since the stride is not constant, these operations are not vector references. In spite of this, they can be vectorized using special machine instructions (e.g. on the Cray Y-MP).

Indirect addressing occurs in solving sparse systems of linear equations (a system is called sparse if most of the matrix elements are zero), and in the FFT algorithm for computing the discrete Fourier transform.

3.1.5.2.4 Scalar Temporary Variables A scalar variable may inhibit vectorization in a loop where all the array references are vector references. Consider, e.g., the code

```
do i=1,n
   s=a(i)+b(i)
   r(i)=s*(x(i)+y(i))
   z(i)=s*s/y(i)
enddo
```

If at each iteration of the loop, the value of `s` is to be stored in a scalar register, then vectorization is not possible. However, by creating a temporary vector, stored in a vector register, the compiler can generate vector instructions:

```
V1=a(1:n)+b(1:n)
r(1:n)=V1*(x(1:n)+y(1:n))
z(1:n)=V1*V1/y(1:n)
s=[last element of V1]
```

3.1.5.2.5 Reduction of a Vector to a Scalar A common difficulty in vectorization is reduction operations where a vector is reduced to a scalar, e.g. summation

```
Fortran 77                  Fortran 90
----------                  ----------
s=0.0
do i=1,n                    s=sum(x(1:n))
   s=s+x(i)
enddo
```

The same type of reduction operation occurs in matrix multiplication

```
do j=1,n
   do i=1,n
      a(i,j)=sum(b(i,:)*c(:,j))
   enddo
enddo
```

If `n` is very large, then the compiler can optimize this operation so that most of the operations are vector instructions.

Example: Consider the summation $\sum_{i=1}^{n} x(i)$, where $n = 1000 * 64$. the computation can be vectorized

```
0 --> V0
do i=1,1000,2
   vload x((i-1)*64+1:i*64) --> V1
   vadd V0 + V1 --> V2
   vload x(i*64+1:(i+1)*64) --> V3
```

```
      vadd V2 + V3 --> V0
   enddo
   Add the elements  of V0 using a  special operation
```

When `n` is less than 64, then reduction operations cannot be vectorized in the same way as ordinary vector operations (that would presuppose that arithmetic operations could be performed with operands in the same vector register). But since this type of operations is so common, many vector computers have special machine instructions for performing them, so that they execute faster than scalar operations, but not quite as fast as ordinary vector instructions.

3.1.5.3 Vectorization Inhibitors

A loop cannot be vectorized if it has

1. recursion
2. a subroutine call
3. I/O operations
4. assigned `goto` statements (†)
5. certain nested `if` blocks
6. `goto` statements that lead out of the loop
7. `goto` into the loop (†)

((†) denotes statements/constructs that no responsible programmer would use anyway.)

3.2 Message passing

Message passing is a programming model mainly used for MIMD computers with distributed memory. Such parallel computers consist of a number of nodes, connected in a network with a certain topology, for example a two-dimensional net or a hypercube. Often the nodes have a processor for computations and a separate processor for communication. The nodes can be identical and tightly coupled (e.g. Cray T3E or IBM SP), or heterogeneous and loosely coupled (e.g. clusters of workstations connected by Ethernet). As there is no shared memory, the processors communicate by sending and receiving messages over the network.

In the programming model there is a group of **processes**, with a certain process topology represented by a **process graph**, e.g. a ring or a two-dimensional net. The processes are assumed to execute asynchronously, and communication is performed by sending and receiving messages. In most cases, the process graph is embedded in the parallel computer so that each process is executed by a separate processor, and so that

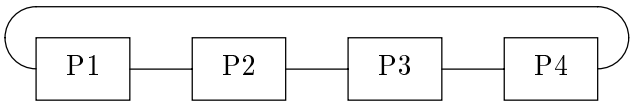


Figure 3.1: Ring of processes, $p = 4$.

neighbors in the process graph are neighbors in the graph of the parallel computer. This is important for load balancing and minimizing the communication time.

The main reason for distinguishing between the *physical processors* and their network topology, and the *logical processes* and their network, is to make the software portable.

In the past each manufacturer of message passing computers supplied a vendor-specific set of communication routines. The first message passing system that was available for a rather wide range of computers was PVM (Parallel Virtual Machine). Since a couple of years, standardization efforts have taken place, which resulted in MPI (Message Passing Interface)[31], a de facto standard for message passing. BLACS (Basic Linear Algebra Communication Subprograms) [10] is a message passing interface designed for the library ScaLAPACK.

Before any communication can take place, a process structure is set up by a call to the communication system. For example, a twodimensional grid of processes is created by

```
gridinit(nprow,npcol)
```

where the process grid is to have dimension `nprow`×`npcol`. Then the individual processes can find their identities, `myid` in this book, and position in the grid. Messages can be sent to other processes by addressing them by their position in the grid, or by sending on a certain link.

In real message passing systems there are a large number of different communication routines, with different variants, e.g., for blocking and non-blocking communication. For our purposes is it sufficient to consider the following primitives with simplified syntax and loosely defined semantics.

`send(destination,data)` Data are sent from the process to another process. The address `destination` can be either a process number or a relative position in a process graph, see below.

`receive(source,data)` Sometimes the `source` of the data is specified, sometimes we assume that the data can be received from any other process.

`broadcast(data)` One process sends the same message to all the others.

If the process graph is a ring, see Figure 3.1, then each process is assumed to know which process is its neighbor to the west and east, and if process P1 executes

```
send(east,data)
```

then `x` is sent to P2. For P2 to access the data sent, it executes

```
receive(east,data)
```

To broadcast data from one process, denoted root, to all the others in a group we use

```
broadcast(data)
```

In some message passing systems it is possible to specify if the broadcast operation shall use a pipelined approach or a tree algorithm, cf. Section 2.6.1.

A very simple example of a message passing algorithm is given below, where a sum $s = \sum_{i=1}^{n} x_i$ is computed by p processes, numbered from 1 to `p`. The vector is distributed by process 1 to all the others, then each process computes its partial sum, and finally the results are sent back to process 1.

Here we used the variant of `send` where a messages is sent to a specific process by specifying its number.

Usually one writes only *one piece of code that is executed by all processes.* Such a codingprinciple is often referred to as *SPMD: Single Program, Multiple Data.*

```
! Process 1 sends a portion of the vector to each other process
! It is assumed that n is an integer times p

ndp=n/p
if myid = 1 then
   do i=2,p
      send(i,x((i-1)*ndp+1:i*ndp))
   enddo
else
   receive(xloc(1:ndp))
endif

! Each process holds its portion of the vector x in
! local variable xloc, and performs the computation

sloc=sum(xloc(1:ndp))

! The result is sent to process 1

if myid = 1 then
   do i=2,p
      receive(s)           ! receive from any process
      sloc=sloc+s
   enddo
else
   send(1,sloc)
endif
```

3.3 Shared memory parallelism – OpenMP

It is probably easiest to describe shared memory parallelism in terms of a particular programming system, and we are going to use OpenMP. This does not mean that the programming model was first invented and implemented in OpenMP. In fact, this model was used for early parallel computers Cray-XMP and Alliant in the mid-eighties, and was called *microtasking*. One important aim of OpenMP is *portability*: it should be possible to execute the same parallel code on any shared memory system.

The purpose of this presentation is not to give a comprehensive overview of OpenMP, but rather to introduce shared memory parallel programming, using some concepts from OpenMP. For a more detailed description, see `www.openmp.org` and [5][2].

In the OpenMP shared memory programming model one usually start out with a sequential code, and then adds *compiler directives* that instruct a parallelizing compiler to introduce parallel execution of certain code segments. The compiler directives have several different forms, the following is typical:

```
!$omp parallel
     code
!$omp end parallel
```

The execution of the code is based on a *fork-join* model, where a *master thread* starts the execution, and then spawns a team of *parallel threads*[3]. A computational problem is

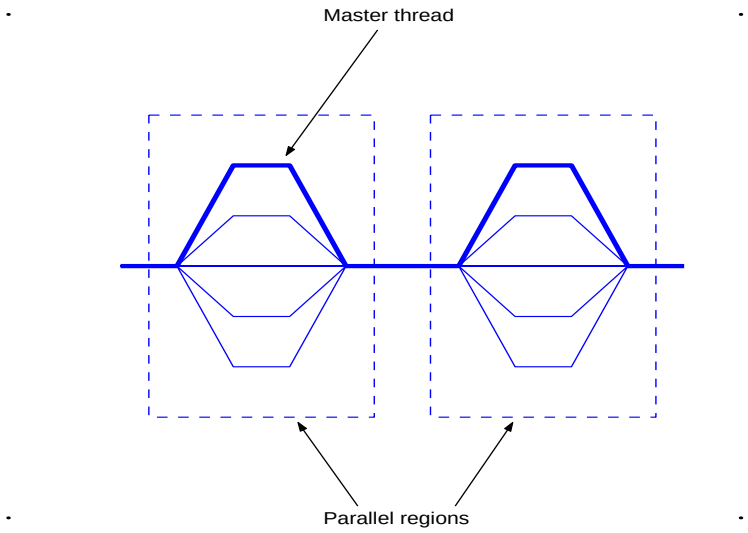


Figure 3.2: Fork-join model for parallel execution.

defined by a set of data, stored in the shared memory, and an algorithm (the code) that

[2]See also Christoph Kessler's homepage `www.ida.liu.se/~chrke/`.

[3]We will mostly use the term *processor* as a synonym of thread.

will lead to the solution. When more than one processor are used to solve a problem, they mostly work on separate pieces of data stored in the shared memory. However, part of the collaboration will involve the same data (otherwise one would have a number of independent problems, which are trivial to parallelize[4]). One important aspect of shared memory parallel programming is to ensure the *integrity of data*:

- The standard type of variables are `shared`; any processor can access them. In order to give different processors their own data for storing partial results, variables can be declared to be `private` to one processor.
- When several processor need to update one single shared variable, care must be taken that they do not access it at the same time. This is done by *synchronization* primitives, *barriers*.

We will take the computation of the sum $s = \sum_{i=1}^{n} x_i$ as a vehicle for discussing the most important aspects of shared memory parallel programming. The following Fortran code performs the computation.

```
s=0
do i=1,n
   s=s+x(i)
enddo
```

In order for a processor to know how much of the computation it should do, it must find out first how many parallel threads there are, and then its own thread number. If the following function is called *in a parallel region*

```
nthr=omp_get_num_threads()
```

then it gives the present number of parallel threads. The assignment

```
myid=omp_get_thread_num()
```

again called *in a parallel region*, gives the thread number (the parallel threads are number from 0 to `p-1`, where `p` is the number of threads). Assuming that `n=nthr*q`, for some positive integer `q`, the following code is a first attempt at parallelizing the summation.

```
! First summation attempt, INCORRECT!
!$omp parallel
   nthr=omp_get_num_threads()
   q=n/nthr                ! Number of terms in each chunk
   myid=omp_get_thread_num()
   first=(myid-1)*q+1
   last=myid*q
   sp=0.                ! Partial sum of each thread
   do i=first,last
```

[4]Such problems are often referred to as *embarrassingly parallel.*

```
      sp=sp+x(i)        ! Sum up my part
   enddo
!$omp end parallel
```

Each processor performs its part of the summation, but the integrity of data has not been taken care of. Since by default each variable is shared (except the loop variable `i`, which by default is private), the same memory locations are used for each processor's `first`, `last`, etc. Obviously, the variables `myid`, `first`, `last` and `sp` must be declared `private` for each participating processor. We must also have a shared variable `s`, to which the different partial sums are to be added. However, it is absolutely necessary that only one processor at a time is allowed to update the variable `s`. By enclosing the updating of the global summation variable in a *critical section*, we ensure that only one processor can access it at a time.

```
! Second summation attempt, CORRECT!
s=0.0
!$omp parallel private(myid,first,last,sp)
   nthr=omp_get_num_threads()
   q=n/nthr             ! Number of terms in each chunk
   myid=omp_get_thread_num()
   first=myid*q+1
   last=(myid+1)*q
   sp=0.0               ! Partial sum
   do i=first,last
      sp=sp+x(i)        ! Sum up my part
   enddo
!$omp critical          ! Only one processor can execute this
   s=s+sp               ! at a time
!$omp end critical
!$omp end parallel
```

Obviously, there is a synchronization point at the end of a parallel region: all processors must have finished there part of the computation before anyone is allowed to continue executing the statement after `!$omp end parallel`. Thus there is an implicit barrier at that point.

There is a more powerful, and simpler, variant, where the iterations are distributed automatically among the processors.

```
! Automatic work-sharing
s=0.0
 !$omp parallel do reduction (+:s)
   do i=1,n
      s=s+x(i)        ! Sum up my part
   enddo
!$omp end parallel do
```

In this `reduction` clause, the variable `s` is assumed to be shared in the enclosing context. The iterations of the loop are distributed among the available processors, and for each copy a private version of `s` is created. The private `s` is used for accumulating a partial sum. At the end of the reduction the value of the global variable is updated to reflect the result of the separate summations. The order of this final summation is undetermined.

It is important to note that codes that involve *recursion*, see page 42, cannot be parallelized, for the same reason that they cannot be vectorized.

3.4 Data parallel programming

Data parallel programming languages were originally designed for SIMD computers but are now used in much more general settings, both for sequential computers and MIMD parallel computers. The following are typical of data parallel programming.

- Data are shared as elements of large, global arrays. Often there is one array element per processor, at least conceptually.
- The parallelism is based on fine-grain operations, usually element-wise operations on arrays.
- Synchronization is implicit, based on language semantics.
- Communication is partially explicit, partially implicit, based on primitive array operations.

In our examples of data parallelism below we will discuss it in terms as it is (was) implemented on a SIMD computer with distributed memory.

Today Fortran 90 is the most widely known example of a data parallel language, but there existed other such languages, e.g., *Lisp (for the Connection Machine), before Fortran 90 was defined. The simplest possible example of a data parallel computation is matrix assignments in Fortran 90.

```
real, dimension(1:m,1:n)       :: a, b, c
....                          ! The matrices are assigned values
a=b*c
c=a-b
```

The statements must be executed in order, so if the code is executed on a parallel computer, there is a synchronization point between each statement. Assuming that each matrix element is stored in a separate processor in such a way that `a(i,j)`, `b(i,j)`, and `c(i,j)` are in the memory of the same processor, we see that the computations can take place simultaneously in all processors, and that no communication is needed.

In the following example communication is needed.

```
real, dimension(1:m)   :: a, b, c
...                           ! Variables are assigned values
```

```
b=(1/sqrt(dot_product(b,b)))*b
                                ! Scale b to have Euclidean
                                ! length 1
c=a+b
```

The code scales the vector `b` to have Euclidean length 1 and adds it to `a`. Again we assume that each element in a vector is stored in a separate processor in such a way that `a(i)`, `b(i)` and `c(i)` are in the memory of the same processor. For the computation of the dot product it is necessary first to multiply the `b` element in each processor by itself. This can be done completely in parallel. Then the products are summed and the division is executed. This requires communication: each processor makes its product available to processor 0, which is assumed to have been assigned the task of summing up the products and performing the division. Typically in an SIMD computer this is done via a register (we call it `breg`), that can be read by the other processors, e.g. in summation operation (denoted `scansum(all.breg)`)[5]. Then the scalar `s` is broadcast to all the processors, and each processor performs the multiplication, and, finally, the addition, in parallel with all the others. The following pseudocode is executed by each processor.

```
! Local variables (scalars): a, b, and c
! Compute b*b and make it available via breg
breg=b*b
if myid=0                   ! Processor 0 performs the summation
   dotp=sqrt(scansum(all.breg))
   all.breg=1/dotp          ! The result is put into the breg of
                            ! each processor
end
b=breg*b                    ! Scale the local component of b
c=a+b                       ! Each processor performs its
                            ! add operation
```

In some cases it is not immediately obvious that a certain array assignment implies communication. Consider, e.g., the slightly modified code.

```
real                    :: s
real, dimension(1:m)    :: a, b, c
...                         ! Variables are assigned values
b=(1/sqrt(dot_product(b,b)))*b
                            ! Scale b to have Euclidean
                            ! length 1
c(1:m-1)=a(2:m)+b(1:m-1)
```

[5]These registers can be considered as a small shared memory used for communication. In an SIMD computer it is relatively easy to ensure the integrity of data in this shared memory, since the same instructions are performed in all processors at the same time (except when a conditional statement implies that only one processor accesses the shared memory).

With the same assumptions as before, we now have more communication: before the final addition is performed the elements `a(2:m)` are sent to the processors holding elements `(1:m-1)`.

It is obvious that in most cases reduction operations, such as summation and dot products, involve communication. In the case of shift operations (see Section 3.1.3) the communication is explicit. Assume that `x` is a twodimensional array, where each element is stored in the local memory of a processor. The processors are assumed to be organized in a twodimensional grid with wrap-around connection (a torus), and each processor can access its neighbour's `breg` by referring to it as `north.breg`, etc. The assignment `y=cshift(x,shift=1,dim=2)`, which is a row-wise shift one position, is then executed in each processor as

```
! local variable (scalar) x
breg=x
x=east.breg
```

In the following sections we will give examples how linear algebra algorithms can be implemented in a data parallel setting.

Chapter 4

Basic Matrix Computations

Linear algebra is at the heart of many scientific computations. It is for this essential reason that most of the important applications developed first are done with linear algebra algorithms.

4.1 Evolution of Linear Algebra Software

Not long ago, if one had to solve a linear system or compute eigenvalues of a small dense matrix, the only possibility was to write one's own code. Then a number of standard packages, notably EISPACK and then LINPACK appeared in the public domain. The idea of using standard software is now not only common but has become almost mandatory because of the need to share expertise in different areas. It is possible to write different variants of the same FORTRAN subroutine which are optimized on different machines. When the user ports a code to another machines, the user will not have to do the additional tuning work that would otherwise have been necessary if the optimized libraries did not exist. The only requirement with the 'common library' approach is the necessity to have standards.

One of the successes in this area is the set of BLAS (Basic Linear Algebra Subroutines) libraries. The BLAS consists of a number of subprograms for basic linear algebra computations. The first level of the BLAS were developed for the LINPACK, which is a library of subroutines for the solution of linear systems of equations.

One of the main reasons for developing the BLAS was to make it easier for the designer of linear algebra programs to write well-structured and efficient code using a set of modules for the most common computations. Another reason was that the BLAS routines can be implemented (often in assembler language) by the different computer manufacturers so that they utilize the hardware as efficient as possible. Thus all machine-dependent details can be hidden inside the BLAS routines, and the programs based on BLAS will be completely portable, i.e., they can be executed on different computers without changes.

The first BLAS subroutines consisted of simple functions such as adding vectors and computing dot products. In many computations, for instance matrix-matrix multiplications, these computations are required in some inner loop. It was soon realized that for

Name	Function	Arguments
–DOT–	dot product	(n,x,incx,y,incy)
–AXPY	y = a*x + y	(n,a,x,incx,y,incy)
–COPY	vector copy	(n,x,incx,y,incy)
–ASUM	sum of absolute values	(n,x,incx)

Table 4.1: Examples of BLAS1 Routines

vector computer with vector registers, it could be a good idea to go at a slightly higher level in the inner loop in order to exploit the presence of the operands in the vector registers. This was essentially known as Level 2 BLAS or BLAS2. More recently, came the emphasis on the use of blocks algorithms in order to exploit data locality as much as possible in computers with hierarchical memory organizations. This gave rise to level 3 BLAS or BLAS3.

There are BLAS routines in single, double, complex single, and complex double precisions.

4.1.1 Level 1 BLAS

The first level of BLAS routines are based on vector–scalar operations and vector–vector operations. The most important are listed in Table 4.1. The – sign in the subprogram names correspond to the precision or version of the function used. For example, the dot-product function has one prefix which can take the values S, D, C, or Z (single, double, complex, double complex) and the suffix U (for the nonhermitian complex inner product $x^T y$), blank, or C (for the Hermitian complex product) $x^H y$.

Among the arguments, n denotes the number of elements to be processed in the vectors x and y, incx and incy denote the increments (strides) in the vectors x and y, respectively. For example, the norm of a row of a matrix can be computed as follows:

```
real a(100,50)
.....
len=SNRM2(50,a(3,1),100)   ! the norm of row 3
```

The BLAS1 is has been partly superseded by the improvements in the FORTRAN language. In FORTRAN 90, a SAXPY operation can be replaced by a statement of the form

```
y(1:n) = y(1:n) + a * x(1:n)  .
```

In fact, even a usual do loop equivalent to the above vector instruction will be appropriately translated into a vector instruction by most compilers with vector processing capability.

4.1.2 Level 2 BLAS

Level 2 BLAS involve matrix-vector operations. In Table 4.2, we list the matrix-vector multiplication, triangular solution routines, and a number of rank one and rank two update

Rootname	Operation	Matrix type
	Matrix Vector Products	
–GEMV	$y := \alpha A^o x + \beta y$	General A
–GBMV	$y := \alpha A^o x + \beta y$	General Banded A
–SYMV	$y := \alpha A x + \beta y$	Symmetric A
–SBMV	$y := \alpha A x + \beta y$	Symmetric Banded A
	Triangular Matrix Vector Products	
–TRMV	$y = T^o x$	General triangular T
–TBMV	$y = T^o x$	Banded triangular T
	Triangular System Solutions	
–TRSV	$y = (T^o)^{-1} x$	General triangular
–TBSV	$y = (T^o)^{-1} x$	Banded triangular
	Rank One and Rank Two Updates	
–GER	$A := \alpha x y^* + A$	General A
–SYR	$A := \alpha x x^T + A$	Symmetric A
–SYR2	$A := \alpha x x^H + \alpha y y^T + A$	Symmetric A

Table 4.2: BLAS2 routines.

routines. In what follows a superscript o denotes either a no-operation ($A^o = A$), the transposition ($A^o = A^T$), or the conjugate transposition ($A^o = A^H$).

A specific rule was used to name the subroutines and the suffixes have the following meaning.

- MV Matrix vector multiplication
- R Rank one update to a matrix
- R2 Rank two update to a matrix
- SV Solving triangular matrix problems.

For example, in matrix–vector multiplication, $y = \alpha Ax + \beta y$, where α and β are scalars, x and y are vectors and A is a matrix. The subroutine is called SGEMV in the case when the matrix is general (i.e., non–symmetric).

4.1.3 Level 3 BLAS

In analogy to Level 1 BLAS for vector–vector operations and Level 2 BLAS for matrix–vector operations, there is Level 3 BLAS for operations of the type matrix-matrix products, rank k updates, and solution of triangular systems with multiple right hand sides. The BLAS3 routines are listed in Table 4.3.

The meaning of the suffix in the naming of the subroutines is as follows.

- MM Matrix - Matrix operations
- RK Rank-k update to a matrix
- R2K Rank-2k update to a matrix
- SM Triangular system solutions with several right-hand-sides.

Rootname	Operation	Matrix type
	Matrix-Matrix multiplications	
–GEMM	$C := \alpha A^o B^o + \beta C$	General case
–SYMM	$C := \alpha AB + \beta C$ or $C := \alpha BA + \beta C$	Symmetric A
	Rank k updates	
–SYRK	$C := \alpha AA^T + \beta C$ or $C := \alpha A^T A + \beta C$	Symmetric A
–HERK	$C := (\alpha AA^H + \beta C)^o$.	Hermitian C
	Triangular matrix A	
–TRMM	$B = \alpha(A^o)B$	Triangular A
–TRSM	$B = \alpha(A^o)^{-1}B$	Triangular A

Table 4.3: BLAS3 routines

It is easy to parallelize Level 3 BLAS routines. Matrix multiplication can be considered as a number of independent matrix–vector multiplications, which can be executed in parallel by different processors. Similarly, columns of the solution of a triangular system with multiple right hand sides are independent and the work can be distributed over multiple processors.

4.1.4 BLAS and Memory Hierarchy

One of the most important conclusions of this section is that in order to write efficient programs on high performance computers, it is necessary to take into account the traffic of operands from primary memory to functional units and back. The following rule should be observed:

> *For each memory reference, perform as many floating point operations as possible.*

We will now consider the different levels of BLAS regarding memory references. In each call of SAXPY, two vectors are loaded and one is stored. Thus $3n$ memory references are made (in the sequel we assume that the vectors have n elements and the matrices have order n). The routine performs n multiplications and n additions, altogether $2n$ flops. The Level 2 BLAS routine SGEMV for matrix-vector multiplication loads a whole matrix (n^2 memory references; we disregard the vectors here). The number of flops is $2n^2$ approximately. Finally, the Level 3 BLAS routine for matrix-matrix multiplication, SGEMM, loads three matrices, stores one, and performs $2n^3$ flops. The results are summarized in Table 4.4.

In dense matrix computations, the ratio of computations over data movement is typically high. For example, multiplying two $n \times n$ matrices requires $2n^3$ floating point operations on $2n^2$ operands, and it produces n^2 result. If we are careful about minimizing data movement, we can increase the efficiency of execution considerably. Blocking, which was at the origin of level 3 BLAS, is precisely geared towards this goal. It is primarily motivated by hierarchical memory systems, but the same idea can be used to minimize communication costs in distributed memory systems.

BLAS level	routine	ref.	flops	flops/ref
1	SAXPY	$3n$	$2n$	$2/3$
2	SGEMV	n^2	$2n^2$	2
3	SGEMM	$4n^2$	$2n^3$	$n/2$

Table 4.4: Memory traffic and floating point operations for BLAS routines.

Next we discuss the problem of multiplying two matrices in some more detail. In particular we consider data movement in a computer with a memory hierarchy, see Section 2.4.1. For simplicity we assume that all matrices involved are square and of dimension n. If the standard formula

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

is used then in the worst case for every element to be computed there will be $2n$ data moves from main memory and one back to memory. This comes to a total of $(2n+1)n^2$ data moves. There are roughly $2n$ data moves for each computation. Consider now a computer in which memory is organized hierarchically. For example, the processor that performs the above computation, may be attached to a very fast local memory, a cache memory. Ideally, we would like all the data in the matrix multiplication to be brought only once in cache. This would entail a total of $3n^2$ data moves from memory, far less than in the previous case. Unfortunately, cache memories are usually small and if the matrix is reasonably large, then this would not be possible.

However, we can certainly fit sub-blocks of the matrix in cache. If we simply divide the matrix into blocks of equal dimension n/q each, then the above formula can be replaced by the block version:

$$C_{ij} = \sum_{k=1}^{q} A_{ik} B_{kj}.$$

Assuming that the cache is large enough to hold three blocks of the matrix, then we we are able to compute $(n/q)^2$ elements by reading $2q$ blocks of $(n/q)^2$ elements each from memory and writing back one block of $(n/q)^2$ results to memory. For each result C_{ij} we therefore need $(2q+1)(n/q)^2$ memory references. The total of data moves from/to memory in this second case is

$$q^2(2q+1)\left(\frac{n}{q}\right)^2 = (2q+1)n^2$$

Notice that the number of moves is no longer cubic with respect to n. Also observe that when $q = n$ we do return to the situation of the non-blocked matrix multiplication involving $(2n+1)n^2$ data moves. This simple illustration shows that data movements from main memory (or for any slow memory which holds the data) can be reduced drastically by a very simple reorganization of the calculation. In [19] it is shown how to exploit this idea to improve performance in dense matrix computations.

BLAS level	routine	operation	Number of processors 1	2	4	8
2	SGEMV	$y := \alpha Ax + \beta y$	311	611	1197	2285
3	SGEMM	$C := \alpha AB + \beta C$	312	623	1247	2425
2	STRMV	$x := Uy$	293	544	898	1613
3	STRMM	$B := UB$	310	620	1240	2425
2	STRSV	$x := U^{-1}x$	272	374	479	584
3	STRSM	$B := U^{-1}B$	309	618	1235	2398
Peak speed			333	666	1332	2664

Table 4.5: Speed (Mflops) of Level 2 and 3 BLAS routines on a Cray Y-MP. All matrices are of order 500. U is upper triangular.

Clearly, it is also possible to exploit blocking in parallel processing. From a processor's point of view, memory is hierarchical, whether it is shared or distributed.

We close this section by giving some statistics (Table 4.5, from [9]) showing that certain BLAS routines from levels 2 and 3 can be implemented to parallelize very well on a shared memory parallel computer. Note that the operation of solving a triangular system of equations is inherently sequential, and does not lend itself to efficient parallelization.

4.2 Matrix – Vector Multiplication

4.2.1 Algorithms for Memory Hierarchies

Assume that we want to compute

$$y = Ax, \tag{4.1}$$

where A has dimension $m \times n$. The components of y are given by

$$y_i = \sum_{j=1}^{n} a_{ij}x_j, \quad i = 1, 2, \ldots, m.$$

There are two immediate versions matrix-vector multiplication algorithms that are based on BLAS 1 routines. The dot-product form consists of computing each component of the resulting vector y as

Fortran 77

```
! SDOT (IJ) version
do i=1,m
   y(i)=0
   do j=1,n
      y(i)=y(i)+a(i,j)*x(j)
   enddo
enddo
```

Fortran 90

```
! SDOT (IJ) version
do i=1,m
   y(i)=dot_product(a(i,1:n),x(1:n))
enddo
```

Symbolically the SDOT version can be represented as

$$\begin{pmatrix} \times \\ \times \\ \times \\ \times \end{pmatrix} = \begin{pmatrix} \leftarrow & - & - & \rightarrow \\ \leftarrow & - & - & \rightarrow \\ \leftarrow & - & - & \rightarrow \\ \leftarrow & - & - & \rightarrow \end{pmatrix} \begin{pmatrix} \uparrow \\ | \\ | \\ \downarrow \end{pmatrix}.$$

This figure should be interpreted as follows: each element of the left hand side vector is equal to the inner product of one row of the matrix A and x. (This type of figures are used in [11]. The presentation in this and the following section is to a considerable extent based on that paper.)

Note that each inner product is independent of others, and therefore all the inner products can be computed in parallel. However, this algorithm has the disadvantage that the elements of A are referenced row-wise, which means that there will occur cash misses.

The second approach is based on the SAXPY operation. By writing the matrix as a collection of column vectors

$$A = (a_{\cdot 1}\, a_{\cdot 2}\, \ldots\, a_{\cdot n}), \qquad a_{\cdot j} = \begin{pmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{mj} \end{pmatrix},$$

and exchanging the order of the loops, the multiplication can be written as

$$y = (a_{\cdot 1}\, a_{\cdot 2}\, \ldots\, a_{\cdot n}) \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \sum_{j=1}^{n} x_j a_{\cdot j}$$

This is done by the code

```
! SAXPY (JI) version
   y(1:m)=0
do j=1,n
     y(1:m)=y(1:m)+a(1:m,j)*x(j)
enddo
```

The `j` loop is a SAXPY operation and symbolically, it can be represented as

$$\begin{pmatrix} \uparrow \\ | \\ | \\ \downarrow \end{pmatrix} = \begin{pmatrix} \uparrow & \uparrow & \uparrow & \uparrow \\ | & | & | & | \\ | & | & | & | \\ \downarrow & \downarrow & \downarrow & \downarrow \end{pmatrix} \begin{pmatrix} \times \\ \times \\ \times \\ \times \end{pmatrix}.$$

Due to the column-wise access pattern, cache misses will not occur, and this variant will execute more efficiently on modern computers with a memory hierarchy.

4.2.1.1 Vector Computers

The situation is very similar for vector computers. In the inner product algorithm, there is a risk of memory bank conflicts, due to the row-wise access of the matrix. Also, even though inner products can be vectorized using special instructions and hardware, they are usually slower than "real" vector operations due to the need to take the summation of the product values. Furthermore, inner products usually entail a higher start-up time. However, if the row dimension is much smaller than the column dimension of A, the dot product form may be advantageous since it involves longer vectors.

In the SAXPY version we have genuine vector operations, which can be chained and executed efficiently. On the other hand, the semantics of Fortran prescribe that at each iteration of the loop, the vector `y` is converted to the floating point format in which it has been declared. Therefore, if, as is often the case, the vector register is wider (has more bits) than the standard word length, then conversion must take place, and this is usually done by storing the vector `y` in primary memory for each iteration of the `j` loop. Since we are only interested in the final value of `y`, there will be $n-1$ unnecessary `vstore` operations.

Instead we would prefer to accumulate `y` in a vector register. This version is sometimes called **GAXPY** (Generalized SAXPY, GAXPY is not one of the BLAS routines). (In the code below we assume that a scalar can be stored in the multiplication unit and used in a vector operation.)

```
! GAXPY version
0 --> V0
do j=1,n,2
   vload a(1:m,j) --> V1
   load x(j) to multiplication unit
   V1*x(j) --> V2
   V2+V0 --> V3
   vload a(1:m,j+1) --> V4
   load x(j+1) to multiplication unit
   V4*x(j+1) --> V5
   V5+V3 --> V0
enddo
vstore V0 --> y(1:m)
```

Unfortunately, compilers do not always recognize when intermediate results could stay in vector registers or when the loads and arithmetic operation can be overlapped. To simulate GAXPY, **loop unrolling** can be done. For simplicity, assume that n is a multiple of 4. The ji version can then be written as

```
! SAXPY version, unrolled loop
y(1:m)=0

do j=1,n,4
   y(1:m)=y(1:m)+a(1:m,j)*x(j)+a(1:m,j+1)*x(j+1)
```

```
            +a(1:m,j+2)*x(j+2)+a(1:m,j+3)*x(j+3)
    enddo
```

Here the compiler can keep the vector `y` in vector registers while four chained vector multiplications and additions are executed, and it will store it in primary memory when `j` changes.

4.2.2 Message Passing Matrix–Vector Multiplication

The message passing matrix–vector multiplication algorithm given below is based on SAXPY operation. We can write

$$y = Ax = \sum_{j=1}^{n} a_{\cdot j} x_j,$$

where $a_{\cdot j}$ denotes the j'th column of A. If the number of processors p is the same as n, and the j'th column $a_{\cdot j}$ of A and the j'th component of x, x_j, are assigned to processor j, then all the products $x_j a_{\cdot j}$ can be computed in parallel.

After all the vectors $x_j a_{\cdot j}$, $1 \leq i \leq n$, are computed, we need to add them up. Of course, this can be done by sending them to one processor that performs all vector additions. However, if the number of elements of y is large, then it may be faster to let the processors share this work. This can be done by a fan-in algorithm, illustrated in Figure 4.1.

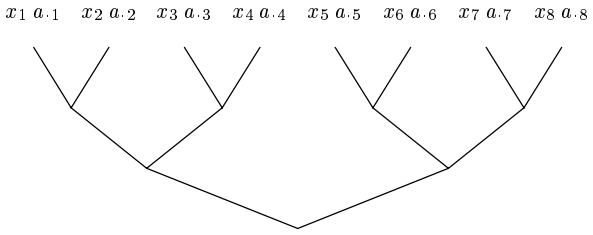


Figure 4.1: Fan-in algorithm for computing $y = Ax = \sum_{j=1}^{n} a_{\cdot j} x_j$.

Assume that $p = n = 2^d$ for a positive integer d, and that the processors are numbered $1, 2, \ldots, p$. Each processor is further assumed to have a local variable, `myid`, which is its identification number. In the code below we use integer division as in Fortran, where, e.g., $1/2 = 0$. Thus

$$(i/2) * 2 = \begin{cases} i-1 & \text{if } i \text{ is odd,} \\ i & \text{if } i \text{ is even.} \end{cases}$$

In the code we use the communication primitives `send(destination,vector)` and `receive(vector)`, where the latter means that a message from any other processor is received. The whole computation is done when each processor executes the following code.

ALGORITHM **4.1 Parallel matrix-vector multiplication. Fan-in algorithm.**

```
! Each processor has local variables aj (vector) and xj (scalar)
! with its column from the matrix A, and its component of the
! vector x, respectively.
! It is assumed that the number of processors is equal to 2**d
! NOTE: "/=" means "not equal"
!
y(1:m)=xj*aj(1:m)
if (myid/2)*2+1 = myid then
   send(myid+1,y)            ! Even processors will continue
else
   do k=1,d                  ! Fan-in algorithm
      if (myid/2**k)*(2**k) = myid then
         receive(y1)
         y=y+y1
         if (myid/2**(k+1))*(2**(k+1)) /= myid and k<d then
            send(myid+2**k,y)
         endif
      endif
   enddo
endif
```

Synchronization is needed to ensure that the necessary multiplications are complete before addition begins and this is performed in a natural way via communication. For example, when P_1 and P_2 are done with multiplication, one of them begins addition with the data it receives from the other but it will wait until the data are received and this wait achieves the necessary synchronization.

It is easy to generalize Algorithm 4.1 to the case when each processor holds a block of columns from the matrix A.

The SDOT (inner product) version of matrix vector multiplication is derived from the expression

$$y = Ax = \begin{pmatrix} a_{1.}^T \\ a_{2.}^T \\ \vdots \\ a_{m.}^T \end{pmatrix} x = \begin{pmatrix} a_{1.}^T x \\ a_{2.}^T x \\ \vdots \\ a_{m.}^T x \end{pmatrix},$$

where the ith row of A is denoted as $a_{i.}^T$. For simplicity we assume that $p = m$ and that processor i has $a_{i.}^T$ and the whole vector x. Then each processor can compute its inner product $a_{i.}^T x$, and achieve perfect parallelism.

4.2.3 Shared Memory Parallel Matrix–Vector Multiplication

Matrix–vector multiplication using the SDOT version is trivial to parallelize using OpenMP, since the computation of each component of the vector y is independent of the others. However, due to the row-wise access, this version of is unsuitable also in the parallel shared

memory context. The SAXPY version can easily be parallelized based on the summation code in Section 3.3.

Algorithm **4.2 Shared memory parallel matrix-vector multiplication.**

```
y(1:m)=0.
!$omp parallel private(myid,first,last,yp)
   nthr=omp_get_num_threads()
   q=n/nthr                ! Number of vectors in each chunk
   myid=omp_get_thread_num()
   first=myid*q+1
   last=(myid+1)*q
   yp(1:m)=0.                ! Partial sum
   do j=first,last
      yp(1:m)=yp(1:m)+a(1:m,j)*x(j)
   enddo
!$omp critical             ! Only one processor can execute
   y(1:m)=y(1:m)+yp(1:m)   ! this at a time
!$omp end critical
!$omp end parallel
```

4.2.4 Data Parallel Matrix–Vector Multiplication

In the formula for the SAXPY based algorithm

$$y = \sum_{j=1}^{n} x_j a_{\cdot j},$$

each element in column j of the matrix A is multiplied by the same quantity, x_j. In Fortran 90, we can express this in a data parallel way by creating a matrix of the same dimensions as A, where all the elements of column j are equal to x_j, and then performing the elementwise matrix multiplication. This is performed using the `spread` function:

```
A*spread(x(1:n),1,m)    ! make m copies of x(1:n) along the
                        ! first dimension, i.e. by columns.
                        ! Multiply by A, element-wise.
```

Then, to form the vector y we sum the elements of the result matrix rowwise:

```
y(1:m) = sum(A*spread(x(1:n),1,m),2)
```

The communication in this algorithm is described in Figure 4.2.

Assume that the elements of A are distributed to processors by blocks, that the elements of x are in the same processors as the first row of A, and, finally, the elements of y are in the same processors as the first column of A. In HPF we can write this

Spread x | Elementwise multiplication | Sum the elements of each row

Figure 4.2: Data parallel matrix-vector multiplication.

```
!HPF$ distribute (block,block)  :: a
!HPF$ align with a(1,:)         :: x(:)
!HPF$ align with a(:,1)         :: y(:)
```

Matrix-vector multiplication often appears as an intermediate step of other larger computation and which algorithm to use will depend on the storage of A and x at the time when the multiplication is required and also what computation follows after.

4.3 Matrix – Matrix Multiplication

Consider the problem of computing the matrix-matrix multiplication

$$C = AB,$$

where, for simplicity we assume that both A and B are $n \times n$. A common matrix multiplication algorithm can be written in Fortran 77 as

```
! SDOT (IJK) version
do i=1,n
   do j=1,n
      c(i,j)=0
         do k=1,n
            c(i,j)=c(i,j)+a(i,k)*b(k,j)
         enddo
   enddo
enddo
```

and in Fortran 90 as

```
! SDOT (IJK) version
do i=1,n
   do j=1,n
      c(i,j) = dot_product(a(i,1:n), b(1:n,j))
   enddo
enddo
```

The above version is based on inner products (SDOT). Similarly as in matrix–vector multiplication, we now have $3! = 6$ different ways by changing the order of execution of the three FORTRAN loops.

Disregarding the zero initialization of A, we can write the generic matrix multiplication algorithm

```
! Generic matrix multiplication
do -----
   do -----
      do -----
         c(i,j)=c(i,j)+a(i,k)*b(k,j)
      enddo
   enddo
enddo
```

We denote the version names by the order of the indices. By making `i` the last index, we have column oriented vector operations in the innermost loop and obtain KJI and JKI versions.

The JIK version comes by switching the j and i loops in IJK version.

```
! JIK version

do j=1,n
   do i=1, n
      c(i,j) = dot_product( a(i,1:n), b(1:n,j) )
   enddo
enddo
```

This results in a slight variation in which the dot product is still used but the elements of C are computed column-wise instead of row-wise.

The JKI version involves column-wise SAXPY operation, i.e. vector operations with stride one.

```
!  JKI version

do j=1,n
   do k=1, n
      c(1:n,j) = c(1:n,j) + a(1:n,k) * b(k,j)
    enddo
enddo
```

The two innermost loops of the JKI version can be illustrated as

$$\text{JKI}: \quad \begin{pmatrix} & \uparrow & \\ & | & \\ & | & \\ & \downarrow & \end{pmatrix} = \begin{pmatrix} \uparrow & \uparrow & \uparrow & \uparrow \\ | & | & | & | \\ | & | & | & | \\ \downarrow & \downarrow & \downarrow & \downarrow \end{pmatrix} \begin{pmatrix} & \times & \\ & \times & \\ & \times & \\ & \times & \end{pmatrix}$$

This version can be implemented using SAXPY operations, but we see that since the computation of each column of C can be finished before it is stored back, we have a GAXPY oriented algorithm. Furthermore, since the columns of A need only be loaded from primary memory, and not stored back, we have only half as much memory traffic as in the KJI version. One interesting point is that the SAXPY operations may involve vectors of different lengths. In the IKJ version, we had vectors of length n and stride n, whereas in the JKI version we have a vector length of n but a unit stride.

The final set of versions consists of starting with the k-loop.

```
!  KJI version

do k=1, n
   do j=1, n
      c(1:n,j) = c(1:n,j) + a(1:n,k) * b(k,j)
   enddo
enddo
```

If we consider the i and j loops together, we discover that each of them corresponds to a rank-one update, i.e., the computation corresponds to the formula

$$C = AB = \begin{pmatrix} a_{\cdot 1} & a_{\cdot 2} & \dots & a_{\cdot n} \end{pmatrix} \begin{pmatrix} b_{1\cdot}^T \\ b_{2\cdot}^T \\ \vdots \\ b_{n\cdot}^T \end{pmatrix} = \sum_{k=1}^{n} a_{\cdot k} b_{k\cdot}^T,$$

where $a_{\cdot k}$ are column vectors of A and $b_{k\cdot}^T$ are row vectors of B, respectively. The matrix $a_{\cdot k} b_{k\cdot}^T$ is called an *outer product matrix,* and the above version is referred to as the outer product form. We can illustrate the computation of the two innermost loops symbolically:

$$\text{KJI}: \quad \begin{pmatrix} \uparrow & \uparrow & \uparrow & \uparrow \\ | & | & | & | \\ | & | & | & | \\ \downarrow & \downarrow & \downarrow & \downarrow \end{pmatrix} = \begin{pmatrix} & & \uparrow & \\ & & | & \\ & & | & \\ & & \downarrow & \end{pmatrix} \begin{pmatrix} & & & \\ \times & \times & \times & \times \\ & & & \end{pmatrix}$$

The figure should be interpreted: "each column of C is updated by adding a multiple of a column from A". We see that each column of C must be fetched from primary memory, updated and then stored back.

The outer product form of matrix multiplication can be written in Fortran 90 using the function `spread`.

```
do k=1, n
  c(1:n,1:n)=c(1:n,1:n)+spread(a(1:n,k),2,n)*spread(b(k,1:n),1,n)
enddo
```

The memory traffic in three of the versions described above, SDOT (IJK), KJI and JKI, is summarized in Table 4.6 (only the highest order term is given).

SDOT	`kji`	`jki`
n^3	$2n^3$	n^3

Table 4.6: Memory traffic in matrix multiplication.

4.3.1 Message Passing Matrix Multiplication

There are several ways of implementing matrix–matrix multiplications on a message passing system. We will present a rather simple method, which executes surprisingly well on modern computers, and actually outperforms other more complicated algorithms [20]. Consider

$$C = AB,$$

where, for simplicity, A, B, and C are assumed to be $n \times n$ matrices. The matrices are distributed by equal size blocks over a square mesh of processes. Thus, with

$$A = \begin{pmatrix} A_{11} & \cdots & A_{1r} \\ \vdots & & \vdots \\ A_{r1} & \cdots & A_{rr} \end{pmatrix},$$

where each A_{ij} is a $k \times k$ matrix with $n = kr$, we assume that A_{ij} is stored in process (i, j). The number of processes is $p = r^2$.

To derive the algorithm we first partition A by block rows and B by block columns:

$$\begin{aligned} A &= \begin{pmatrix} A_{1\cdot} \\ \vdots \\ A_{r\cdot} \end{pmatrix}, \quad A_{i\cdot} = \begin{pmatrix} A_{i1} & \cdots & A_{ir} \end{pmatrix} \\ B &= \begin{pmatrix} B_{\cdot 1} & \cdots & B_{\cdot r} \end{pmatrix}, \quad B_{\cdot j} = \begin{pmatrix} B_{1j} \\ \vdots \\ B_{rj} \end{pmatrix}. \end{aligned}$$

With the assumed distribution of the matrix, the whole block $A_{i\cdot}$ is stored on *process row* i, and, similarly, $B_{\cdot j}$ is stored in *process column* j. Using this notation we see that the block C_{ij} is given by

$$C_{ij} = A_{i\cdot}B_{\cdot j} = \sum_{k=1}^{n} A_{ik}B_{kj}.$$

This can be written in outer product form,

$$C_{ij} = \sum_{l=1}^{n} a_i^l (b_l^j)^T,$$

where a_i^l denote the columns of $A_{1\cdot}$, and $(b_l^j)^T$ the rows of $B_{\cdot j}$, respectively,

$$A_{i\cdot} = \begin{pmatrix} a_i^1 & \cdots & a_i^n \end{pmatrix}, \quad B_{\cdot j} = \begin{pmatrix} (b_1^j)^T \\ \vdots \\ (b_n^j)^T \end{pmatrix}.$$

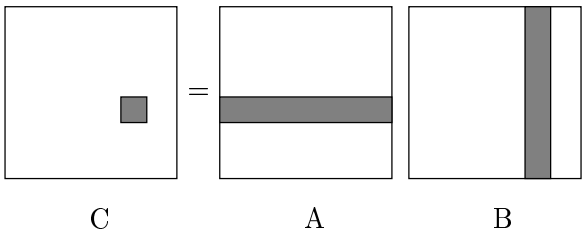


Figure 4.3: The shaded areas of A and B are used in the computation of C_{ij}.

We illustrate the data used in the computation of a block C_{ij} in Figure 4.3. Now, each process in row i will need all the column vectors a_i^l in turn for its sum, and each process in column j will need each row vector $(b_l^j)^T$ in turn. Thus, process (i,j) can compute its block of C by executing the following pseudo-code:

ALGORITHM **4.3 Message Passing Matrix Multiplication**

$C_{ij} = 0$

`do l=1,n`

`if I hold` a_i^l `then broadcast it within my process row`

`if I hold` b_l^j `then broadcast it within my process column`

$C_{ij} = C_{ij} + a_i^l (b_l^j)^T$

`enddo`

It is now straightforward to derive a measure of the efficiency of this algorithm. Assuming that a floating point operation (addition or multiplication) takes γ seconds, we see that the computations in one step of the algorithm takes $2k^2\gamma$ seconds (both a_i^l and b_l^j have k elements), and the total time is

$$2nk^2\,\gamma = 2n\frac{n^2}{r^2}\,\gamma = \frac{2n^3}{p}\,\gamma.$$

To estimate the communication time, we assume that sending a message of size m from one processor to any other takes $\tau + \beta m$, and that a broadcast over r processes takes $\log_2 r(\tau + \beta m)$, see Section 2.6.1. The communication in each step takes $2\log_2 r(\tau + \beta k)$, since k elements are broadcast row-wise and column-wise[1]. Thus the total communication time is

$$2n\log_2 r(\tau + \beta k) = 2n\log_2 r\left(\tau + \frac{\beta n}{r}\right) = n\log_2 p\left(\tau + \frac{\beta n}{\sqrt{p}}\right),$$

where we have used $r^2 = p$.

[1]Actually, some processes will finish one step and begin with the next before other processes have completed the step. In the end, however, all processes will have to wait until all other have finished their work. Therefore it is adequate to assume that all steps take as long for all processes.

To demonstrate the efficiency of the algorithm we compute the speed-up (i.e. the time for the sequential algorithm over the time for the parallel algorithm)

$$S(n,p) = \frac{2n^3\gamma}{2n^3\gamma/p + n\log_2 p(\tau + \beta n/\sqrt{p})} = \frac{p}{1 + \frac{p\log_2 p}{2n^2}\frac{\tau}{\gamma} + \frac{\sqrt{p}\log_2 p}{2n}\frac{\beta}{\gamma}}.$$

The efficiency is

$$E(n,p) = \frac{S(n,p)}{p} = \frac{1}{1 + \frac{p\log_2 p}{2n^2}\frac{\tau}{\gamma} + \frac{\sqrt{p}\log_2 p}{2n}\frac{\beta}{\gamma}}.$$

If we ignore the $\log_2 p$ factor, which grows slowly for large p, we see that the algorithm is *scalable* in the following sense: The efficiency stays the same when we increase the number of processors, if at the same time we let the problem size (matrix dimension n) grow as $n = C\sqrt{p}$. The concept of scalability will be further discussed in Section 4.4.

The algorithm can be made even more efficient if we do not perform a logarithmic broadcast, but instead consider the array of processes as a two-dimensional pipeline. Here the process that holds the current column vector a_i^l sends it to the nearest neighbor to the east, which sends it to the next neighbor, and so on. The vectors b_l^i are send to the neighbors to the south.

ALGORITHM **4.4 Pipelined Message Passing Matrix Multiplication**

$C_{ij} = 0$

```
do l=1,n
     if I hold a_i^l then
          send(east,a_i^l,r-1)
     else
          receive(west,a_i^l,count)
          if count>1 send(east,a_i^l,count-1)
     endif
     if I hold b_l^j then
          send(south,b_l^j,r-1)
     else
          receive(north,b_l^j,count)
          if count>1 send(south,b_l^j,count-1)
     endif
     C_ij = C_ij + a_i^l (b_l^j)^T
enddo
```

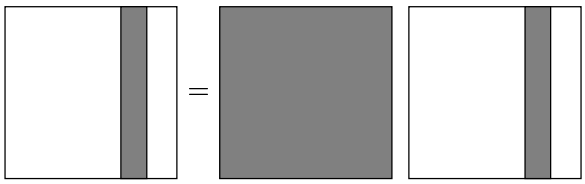


Figure 4.4: The shaded areas of A and B are used in the computation of a block column of C.

For the timing analysis we first note that since we are performing a broadcast using the "linear" algorithm, it takes

$$2(r-1)(\tau+\beta k) \approx 2\sqrt{p}\,(\tau+\beta\frac{n}{\sqrt{p}}) \tag{4.2}$$

before the last process in the pipeline receives its first data. After that all processes are busy receiving and sending data, and doing their updates of C_{ij}. There are n steps in the algorithm, so this takes approximately

$$n(2k^2\gamma+2(\tau+\beta k)) = \frac{2n^3}{p}\gamma + 2n(\tau+\beta\,\frac{n}{\sqrt{p}}).$$

At the end of the computation, the process that sends the last message will be idle while the others are finishing up. This takes about as long as the initial wait (4.2). Thus the total time is

$$\frac{2n^3}{p}\gamma + 2n(\tau+\beta\,\frac{n}{\sqrt{p}}) + 4\sqrt{p}\,(\tau+\beta\frac{n}{\sqrt{p}}), \tag{4.3}$$

approximately. Note that the $\log_2 p$ factors have disappeared. In the same way as before, we now get the efficiency

$$E_b = E_b(n,p) \approx \frac{1}{1+\frac{p}{n^2}\frac{\tau}{\gamma}+\frac{\sqrt{p}}{n}\frac{\beta}{\gamma}}, \tag{4.4}$$

where, for simplicity, we have disregarded the less important last term in (4.3). Thus, the above pipelined message passing matrix multiplication algorithm is scalable in the same sense as the previous version.

It is also possible to derive a version of this algorithm, where data are distributed by block columns. In Figure 4.4 we illustrate the way A and B are accessed to compute one block column in C. It is natural to implement this algorithm on a ring of processes. Since the process that holds a certain block of C, also holds the corresponding block of B, it is only necessary to send the columns of A around the ring. This can be done either by a logarithmic broadcast or in a pipelined fashion.

It is rather straightforward to repeat the timing analysis above for this case. The total time for the pipelined block column version becomes

$$\frac{2n^3}{p}\gamma + n(\tau+\beta n) + 2p(\tau+\beta n),$$

approximately, where the last term is the time for the startup and final phases (when the pipeline is filling up and when the final message is sent around). Simplifying by ignoring the last term, we get the efficiency

$$E_{bc} = E_{bc}(n,p) \approx \frac{1}{1 + \frac{p}{2n^2}\frac{\tau}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}}. \tag{4.5}$$

Note that here we have a term $\frac{p}{2n}\frac{\beta}{\gamma}$ in the denominator, as opposed to (4.4), where instead we have $\frac{p}{n^2}\frac{\tau}{\gamma} + \frac{\sqrt{p}}{n}\frac{\beta}{\gamma}$. This will be important in the analysis of the scalability of these two variants in Section 4.4.

4.3.2 Shared Memory Parallel Matrix Multiplication

Using the code for matrix–vector multiplication given in Section 4.2.3 it is relatively easy to parallelize a SAXPY version of matrix multiplication.

4.3.3 Data Parallel Matrix Multiplication

Matrix multiplication appears to be ideally suited for data-parallel computation because of the regular nature of the operation. Our description of Cannon's algorithm for multiplication of square matrices will also show that the communication aspects of data-parallel computations are very important.

Consider

$$C = AB,$$

where A, B, and C are square matrices. Assume, for the moment, that they all have order 4, and that we have a 4×4 array of processors. In the algorithm, processor (i,j) will compute c_{ij}. From the definition of matrix multiplication, the elements of the first column of C are given by

$$\begin{aligned}
c_{11} &= \sum_{k=1}^{4} a_{1k}b_{k1} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41} \\
c_{21} &= \sum_{k=1}^{4} a_{2k}b_{k1} = a_{22}b_{21} + a_{23}b_{31} + a_{24}b_{41} + a_{21}b_{11} \\
c_{31} &= \sum_{k=1}^{4} a_{3k}b_{k1} = a_{33}b_{31} + a_{34}b_{41} + a_{31}b_{11} + a_{32}b_{21} \\
c_{41} &= \sum_{k=1}^{4} a_{4k}b_{k1} = a_{44}b_{41} + a_{41}b_{11} + a_{42}b_{21} + a_{43}b_{31}
\end{aligned}$$

Note that we have written the sums in a nonstandard order. If at the start of the computations b_{i1} and a_{ii} are in processors $(i,1)$, $i = 1, \ldots, 4$, then the first term in each equation can be computed in parallel.

The elements of the second column of C are given by

$$\begin{aligned}
c_{12} &= \sum_{k=1}^{4} a_{1k}b_{k2} = a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} + a_{14}b_{42} \\
c_{22} &= \sum_{k=1}^{4} a_{2k}b_{k2} = a_{22}b_{22} + a_{23}b_{32} + a_{24}b_{42} + a_{21}b_{12} \\
c_{32} &= \sum_{k=1}^{4} a_{3k}b_{k2} = a_{33}b_{32} + a_{34}b_{42} + a_{31}b_{12} + a_{32}b_{22} \\
c_{42} &= \sum_{k=1}^{4} a_{4k}b_{k2} = a_{44}b_{42} + a_{41}b_{12} + a_{42}b_{22} + a_{43}b_{32}
\end{aligned}$$

Similarly, if at the start of the computations b_{i2} and $a_{i,i+1}$ are in processors $(i,1)$, $i = 1,\ldots,4$, then the second term in each equation can be computed in parallel, and, in addition, this can be done in parallel with the computations for the first column in C.

Thus, with the initial distribution of matrix elements to processors illustrated in Figure 4.5(a), each processor can compute a first term in its sum.

To continue the computation, the elements of A must be shifted horizontally and the elements of B vertically (with wrap-around, i.e., an element at the bottom of the array is moved to the top, and correspondingly for horizontal shift). After the first shift the elements of A and B are as in Figure 4.5(b), and a second term in each sum can be computed. The algorithm proceeds as illustrated in Figure 4.5(c) and (d), and after 4 elementwise multiplications and 3 shifts, the result C is computed.

The algorithm is implemented by the following Fortran 90 subroutine.

```
subroutine cannon(a,b,c,n)
integer                          :: n
real, dimension(1:n,1:n)         :: a,b,c

!local variable
integer                          :: k

! Perform the initial skewing of a and b
a=cshift(a,(/(k, k=0,n-1)/),2)      ! Horizontal
b=cshift(b,(/(k, k=0,n-1)/),1)      ! Vertical
c=0.0
do k=1,n
   c=c+a*b                  ! Elementwise multiplication
   a=cshift(a,1,2)         ! Shift one step to the left
   b=cshift(b,1,1)         ! Shift one step up
enddo
return
end
```

$a_{11}b_{11}$	$a_{12}b_{22}$	$a_{13}b_{33}$	$a_{14}b_{44}$
$a_{22}b_{21}$	$a_{23}b_{32}$	$a_{24}b_{43}$	$a_{21}b_{14}$
$a_{33}b_{31}$	$a_{34}b_{42}$	$a_{31}b_{13}$	$a_{32}b_{24}$
$a_{44}b_{41}$	$a_{41}b_{12}$	$a_{42}b_{23}$	$a_{43}b_{34}$

(a) Initial alignment

$a_{12}b_{21}$	$a_{13}b_{32}$	$a_{14}b_{43}$	$a_{11}b_{14}$
$a_{23}b_{31}$	$a_{24}b_{42}$	$a_{21}b_{13}$	$a_{22}b_{24}$
$a_{34}b_{41}$	$a_{31}b_{12}$	$a_{32}b_{23}$	$a_{33}b_{34}$
$a_{41}b_{11}$	$a_{42}b_{22}$	$a_{43}b_{33}$	$a_{44}b_{44}$

(b) After first shift

$a_{13}b_{31}$	$a_{14}b_{42}$	$a_{11}b_{13}$	$a_{12}b_{24}$
$a_{24}b_{41}$	$a_{21}b_{12}$	$a_{22}b_{23}$	$a_{23}b_{34}$
$a_{31}b_{11}$	$a_{32}b_{22}$	$a_{33}b_{33}$	$a_{34}b_{44}$
$a_{42}b_{21}$	$a_{43}b_{32}$	$a_{44}b_{43}$	$a_{41}b_{14}$

(c) After second shift

$a_{14}b_{41}$	$a_{11}b_{12}$	$a_{12}b_{23}$	$a_{13}b_{34}$
$a_{21}b_{11}$	$a_{22}b_{22}$	$a_{23}b_{33}$	$a_{24}b_{44}$
$a_{32}b_{21}$	$a_{33}b_{32}$	$a_{34}b_{43}$	$a_{31}b_{14}$
$a_{43}b_{31}$	$a_{44}b_{42}$	$a_{41}b_{13}$	$a_{42}b_{24}$

(d) After third shift

Figure 4.5: Four steps of Cannon's algorithm on 4×4 processors.

The Fortran 90 intrinsic function `cshift` (*circular shift*) performs the communication necessary for this algorithm. The statement `a=cshift(a,1,2)`, specifies that a is to be shifted one step along the second dimension, i.e., rowwise, to the left in wrap-around fasion. In the statement `a=cshift(a,(/(k,k=0,n-1)/),2)`, each row of a is shifted by a different amount, given by the vector `0,1,...,k-1`. This means that the first row is shifted 0 position, the second row 1 position, to the left, etc.

It is straightforward to generalize this algorithm to the case when we have distributed the matrix by blocks to a square $r \times r$ mesh of processors (with wrap-around connections) as in Section 4.3.1. Thus, there are r steps, and in each step the (i, j) processor performs a matrix multiply

$$C_{ij} := C_{ij} + A_{ik}B_{kj},$$

where the matrices have dimension $k \times k = (n/r) \times (n/r)$. Thus the total time for doing the arithmetic is

$$2rk^3\gamma = \frac{2n^3}{r^2}\gamma = \frac{2n^3}{p}\gamma,$$

where we have used $p = r^2$. In each step every processor sends and receives two blocks, so the total communication time (excluding the time for the initial skewing of the matrices) is

$$2r(\tau + \beta k^2) = 2\sqrt{p}\,\tau + 2\frac{n^2}{\sqrt{p}}\,\beta.$$

If the initial skewing is performed by sending matrix blocks to neighbors, then it takes (the last row and column take the longest time)

$$2(r-1)(\tau + \beta k^2) \approx 2\sqrt{p}\,\tau + 2\frac{n^2}{\sqrt{p}}\,\beta.$$

Thus the total time for the block version of Cannon's algorithm is

$$\frac{2n^3}{p}\gamma + 4\sqrt{p}\,\tau + 4\frac{n^2}{\sqrt{p}}\,\beta,$$

approximately. The efficiency becomes

$$E_{\text{cannon}} \approx \frac{1}{1 + 2\left(\frac{\sqrt{p}}{n}\right)^3 \frac{\tau}{\gamma} + 2\frac{\sqrt{p}}{n}\frac{\beta}{\gamma}}. \tag{4.6}$$

4.4 Scalable Computations

Using the example of message passing matrix multiplication given in Section 4.3.1, we will now discuss in some more detail the concept of *scalability*. In [12], the following definition of scalability is given:

> A parallel algorithm is called *highly scalable* if the concurrent efficiency depends on the problem size (number of data) and the number of processors only through their ratio.

Since in matrix computations the problem size is proportional to n^2, where n is the matrix dimension, the definition can be rephrased: A parallel algorithm is called highly scalable if the efficiency depends on n^2/p, but not on p or n separately. The quantity n^2/p is the memory requirement per processor. Therefore, we can also say that an parallel algorithm is highly scalable if the efficiency stays the same when we simultaneously increase the number of processors and the problem size in such a way that the memory requirement per processor is constant.

To illustrate the concept of scalability, we first consider the message passing matrix multiplication algorithm in the case of a ring of processes, where the matrix is distributed by block columns (Section 4.3.1. The efficiency of that algorithm is (see (4.5))

$$E_{bc} = E_{bc}(n,p) \approx \frac{1}{1 + \frac{p}{2n^2}\frac{\tau}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}}.$$

We see that the efficiency depends on p/n, so the algorithm is *not* scalable according to the definition. In order to maintain efficiency in this algorithm when p is increased, we would have to increase n at the same rate. Thus, when we double the number of processors, the memory requirements per processors also doubles. Eventually, when the number of processors is increased, we may run out of memory.

On the other hand, the corresponding algorithm with a square mesh of processes, where the matrix is distributed by block, has efficiency (4.4)

$$E_b = E_b(n,p) \approx \frac{1}{1 + \frac{p}{n^2}\frac{\tau}{\gamma} + \frac{\sqrt{p}}{n}\frac{\beta}{\gamma}}.$$

This algorithm is scalable. In Figure 4.6 we illustrate the deterioration of efficiency in the block-column variant of the algorithm, as the number of processors is increased, and at the same time, the memory per processor is kept constant. The values of the parameters γ, τ, and β are representative of modern (1994) parallel computers [20]. Analogous and similar graphs are given in [20] illustrating runs on actual computers.

We see that Cannon's matrix multiplication algorithm with efficiency (4.6) is also scalable, if the matices are distributed by block over a two-dimensional mesh of processors with wrap-around connections.

Different aspects of scalability are discussed at length in [26].

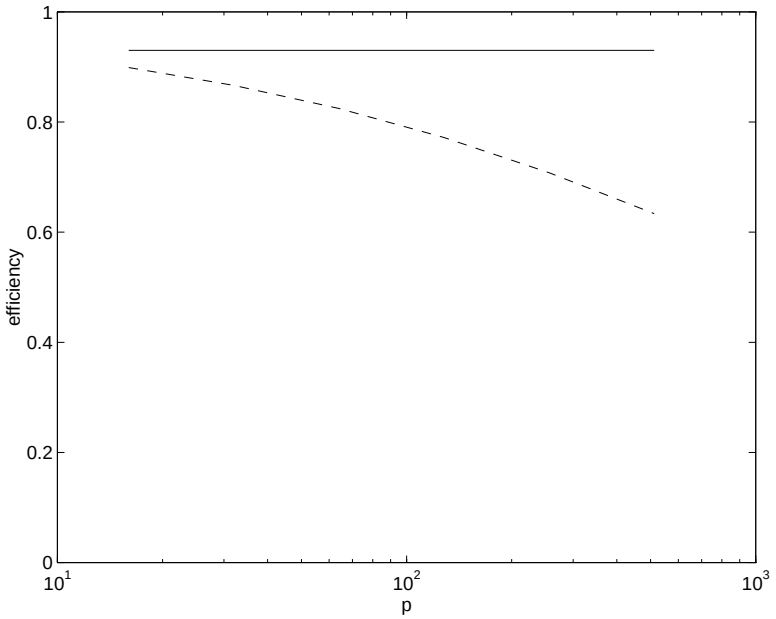


Figure 4.6: Efficiency of the block and block column variants of the message passing matrix multiplication algorithms, solid and dashed curves, respectively, as functions of the number of processors p. The amount of memory per processor is kept fixed. We used $\gamma = 10^{-7}$, $\tau = 10^{-4}$, and $\beta = 10^{-6}$.

Chapter 5

Solution of Dense Linear Systems

Solving linear systems of equations is one of the most important tasks in Scientific Computing. We will devote this chapter to the discussion of algorithms for solving such problems. As will be seen, there is a very rich set of possible algorithms to suit all possible architectures. Thus, the straightforward Gaussian elimination gives rise to several different implementations. We will attempt to describe the algorithms with as much generality as possible with regards to the underlying architecture on which they are executed. An algorithm which is initially designed for a distributed memory computer can very well be implemented on a shared memory computer. This leads to a fundamental distinction between the algorithm and its implementation.

5.1 Gaussian Elimination and LU Decomposition

In this section we describe Gaussian elimination of a dense $n \times n$ matrix A for solving the linear system

$$Ax = b. \tag{5.1}$$

For simplicity of presentation, we assume that no pivoting is required. We describe algorithms for reducing the matrix A to upper triangular form, without performing any operations on the right hand side b. The reduction is implicitly an LU decomposition of A,

$$A = LU,$$

where L and U are lower and upper triangular, respectively. The amount of work to reduce A to upper triangular from is about $2n^3/3$ flops. To solve the linear system (5.1), one can either use the LU decomposition and obtain x by forward and backward substitution. This requires $2n^2$ flops approximately, so it is much cheaper than the reduction to triangular form. Alternatively, one can adjoin b to A and apply all operations to it, as if it were just another column of A. After the reduction, the solution is then obtained by back substitution.

The usual way of deriving the Gaussian Elimination algorithm without pivoting is as

follows. Suppose after $k-1$ steps in the algorithm, the matrix is reduced to the form

$$\begin{pmatrix} a_{11} & a_{12} & & & \dots & & a_{1n} \\ & a_{22} & & & \dots & & a_{2n} \\ & & \ddots & & & & \vdots \\ & & & a_{kk} & a_{k,k+1} & \dots & a_{kn} \\ & & & \vdots & \vdots & & \vdots \\ & & & a_{ik} & a_{i,k+1} & \dots & a_{in} \\ & & & \vdots & \vdots & & \vdots \\ & & & a_{nk} & a_{n,k+1} & \dots & a_{nn} \end{pmatrix}.$$

In the next step of the elimination the elements below the main diagonal in column k will be annihilated. This is done by adding suitable multiples of row k to rows $k+1$ to n. The result is

$$\begin{pmatrix} a_{11} & a_{12} & & & \dots & & a_{1n} \\ & a_{22} & & & \dots & & a_{2n} \\ & & \ddots & & & & \vdots \\ & & & a_{kk} & a_{k,k+1} & \dots & a_{kn} \\ & & & \vdots & \vdots & & \vdots \\ & & & 0 & a'_{i,k+1} & \dots & a'_{in} \\ & & & \vdots & \vdots & & \vdots \\ & & & 0 & a'_{n,k+1} & \dots & a'_{nn} \end{pmatrix},$$

where the transformed elements are

$$a'_{ij} = a_{ij} - a_{ik}a_{kj}/a_{kk}, \qquad j = k+1, \dots, n, \quad i = k+1, \dots, n. \tag{5.2}$$

The multiplier a_{ik}/a_{kk} is saved in the position (i,k) in the matrix.

$$a_{ik} := \frac{a_{ik}}{a_{kk}}, \qquad i = k+1, \dots, n. \tag{5.3}$$

When the reduction is finished, the elements of the lower triangular factor L are those below the diagonal in the array for A. The elements above the diagonal are those of U.

Gaussian elimination without pivoting may be unstable [21, p. 110] (except for special matrices, e.g. symmetric, positive definite or diagonally dominant matrices). In *partial pivoting* the pivot column (in the example column k, the elements below the diagonal) is searched for the element of largest modulus. If that element is found in row i', say, then rows k and i' are swapped (also the rows of part of the lower triangular factor computed so far must be swapped). Then the operations of step k are carried out as described above.

5.2 LU decomposition on Vector Computers

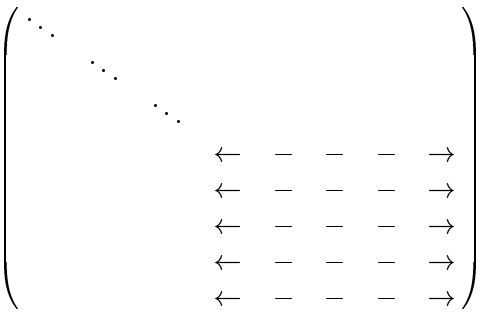
In this section we describe vector implementations of Gaussian elimination, and we also discuss the modifications necessary to take advantage of a memory hierarchy.

The algorithm described in the previous section can be represented in the KIJ variant.

ALGORITHM **5.1 LU decomposition – Row (KIJ) Variant**

```
do k=1,n-1
   do i=k+1,n
      a(i,k)=a(i,k)/a(k,k)
      do j=k+1, n
         a(i,j)=a(i,j)-a(i,k)*a(k,j)
      enddo
   enddo
enddo
```

Here the matrix is referenced as follows in the two innermost loops:

KIJ : 

As in matrix multiplication, we can change the order of the loop and describe Gaussian elimination by the following generic algorithm:

```
do ------
   do ------
      do ------
         a(i,j)=a(i,j)-a(i,k)*a(k,j)/a(k,k)
      enddo
   enddo
enddo
```

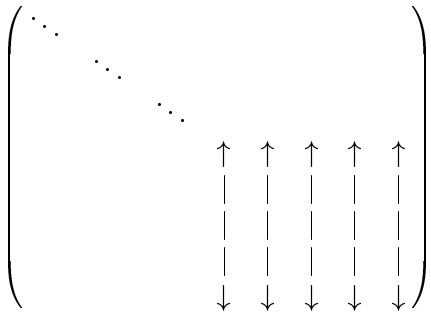
We can permute the loop variables `i`, `j` and `k` in $3! = 6$ different ways. We will discuss how the efficiency of couple of variants depends on the architecture. A column variant of the algorithm is often preferred as it involves vector combinations with stride one. Instead of writing the innermost loop (the `i` loop) explicitly, we express it as a vector statement.

ALGORITHM **5.2 LU decomposition – Column (KJI) variant**

```
do k=1,n-1
   a(k+1:n,k)=(1/a(k,k))*a(k+1:n,k)
   do j=k+1, n
      a(k+1:n,j)=a(k+1:n,j)-a(k,j)*a(k+1:n,k)
   enddo
enddo
```

As in the case of matrix multiplication, the KJI variant is a SAXPY oriented algorithm:

KJI :



Another name for these KIJ and KJI variants is **right-looking** variant, since in each step the matrix elements to the right of the column that is annihilated in the present transformation are referenced (updated).

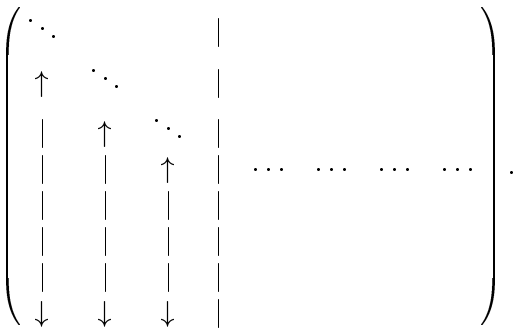
A Gaxpy-oriented variant is obtained by exchanging the `k` and `j` loops:

ALGORITHM **5.3 LU decomposition – JKI variant**

```
do j=1,n
   do k=1,j-1
      a(k+1:n,j)=a(k+1:n,j)-a(k,j)*a(k+1:n,k)
   enddo
   if (j<n) then
      a(j+1:n,j)=(1/a(j,j))*a(j+1:n,j)
   endif
enddo
```

This Gaxpy variant references the matrix in the following way:

JKI :



The rightmost marked column is not modified until the present step of the algorithm. After all previous transformations are applied to that column, the elements below the main diagonal are annihilated, in principle. This need not be done explicitly. The computation actually performed is to divide the elements below the diagonal by the diagonal element, the pivot element. This variant is often referred to as **left-looking**.

As in the matrix multiplication, the most efficient variant on vector computers, e.g., Cray Y-MP, is the one with the minimum number of memory references (all have the same

Variant	KIJ	KJI	JKI
Memory references	$2n^3/3$	$2n^3/3$	$n^3/3$

Table 5.1: Memory references in three variants of the LU decomposition algorithm.

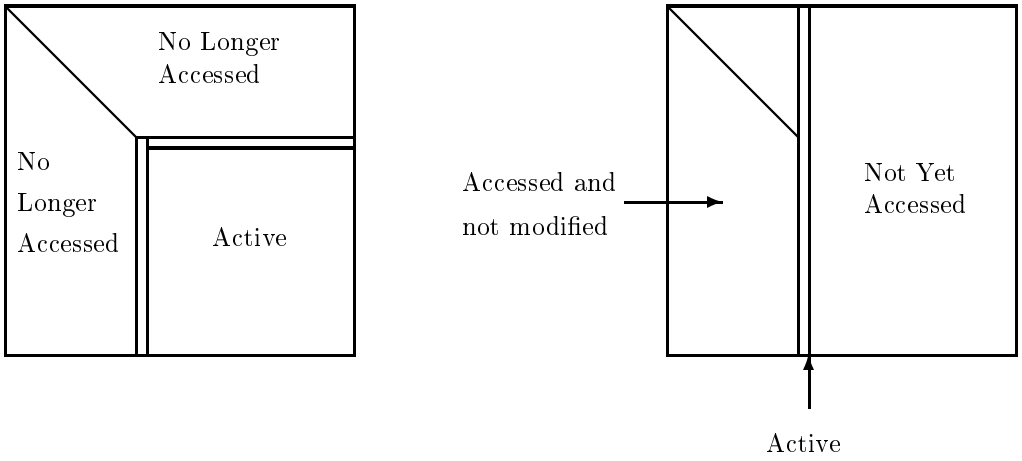


Figure 5.1: Access patterns of the KJI (right-looking), and JKI (left-looking) variants of the LU decomposition algorithm.

number of flops). In Table 5.1 we summarize the number of memory references for the three variants considered. The access pattern of the different versions is shown in Figure 5.1.

5.3 Block Algorithms for Memory Hierarchies

One can maximize the number of flops per memory reference by organizing the computations in block forms. Suppose the matrix A is partitioned into blocks as

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

where A_{11} and A_{22} are square (but not of the same dimension; usually the dimension of A_{11} is much smaller than that of A_{22}). Consider the identity

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & S_{22} \end{pmatrix},$$

where the blocks in L and U have the same dimension as the corresponding blocks in A, and I is the identity matrix. Then multiplying the blocks, we obtain

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & S_{22} + L_{21}U_{12} \end{pmatrix}.$$

Since $L_{11}U_{11} = A_{11}$, we can compute L_{11} and U_{11} by the usual LU decomposition algorithm applied to A_{11}. Then from $L_{11}U_{12} = A_{12}$ and $L_{21}U_{11} = A_{21}$, we can compute U_{12} and L_{21} by solving triangular stystems with multiple right hand sides. Then $S_{22} = A_{22} - L_{21}U_{12}$ can be computed and S_{22} can be stored in the place where A_{22} was stored. The algorithm is summarized:

1. $L_{11}U_{11} = A_{11}$ (Un-blocked LU)
2. $L_{11}U_{12} = A_{12}$ (Triangular solve)
3. $L_{21}U_{11} = A_{21}$ (Triangular solve)
4. $A_{22} := A_{22} - L_{21}U_{12}$ (Matrix multiplication)

Then A_{22} is partitioned into blocks and the analogous procedure is repeated.

For simplicity, in the Algorithm 5.4, we assume that the matrix order n satisfies $n = t * r$, for some integers t and r. The memory reference pattern is illustrated in Figure 5.2.

ALGORITHM **5.4 Right-looking Block Variant**

```
* Right-looking block LU decomposition
do i=1,t
     s=(i-1)*r +1            *Start position of block to decompose
     e=i*r                    *End position of block to decompose
     u=e+1                    *Start position for update
     L_{s:e,s:e}U_{s:e,s:e} = A_{s:e,s:e}      *Un-blocked LU
     U_{s:e,u:n} = L^{-1}_{s:e,s:e}A_{s:e,u:n}  *BLAS-3 routine STRSM
     L_{u:n,s:e} = A_{u:n,s:e}U^{-1}_{s:e,s:e}  *BLAS-3 routine STRSM
     A_{u:n,u:n} = A_{u:n,u:n} - L_{u:n,s:e}U_{s:e,u:n} *BLAS-3 routine SGEMM
enddo
```

A left-looking variant can be derived as follows. Partition the matrices A, L and U as

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & U_{33} \end{pmatrix}, \tag{5.4}$$

and assume that we know U_{11}, L_{11}, L_{21} and L_{31}, and we want to compute L_{22}, L_{32}, U_{12} and U_{22}. By identifying the second column block in A with the second column block of the product in (5.4), we obtain

$$\begin{aligned} A_{12} &= L_{11}U_{12}, \\ A_{22} &= L_{21}U_{12} + L_{22}U_{22}, \\ A_{32} &= L_{31}U_{12} + L_{32}U_{22}. \end{aligned}$$

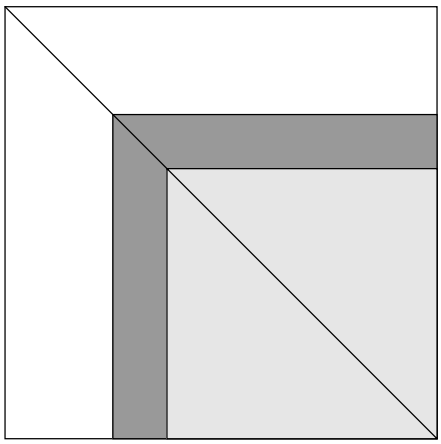


Figure 5.2: Right-looking block LU decomposition. The darker shade indicates the elements that are computed in the present block transformation and the lighter shade indicates elements that are updated.

From the first equation, we compute U_{12} by solving a triangular system

$$L_{11}U_{12} = A_{12}.$$

Then we update the (2,2) and (3,2) blocks in A:

$$\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} := \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} U_{12}.$$

Now we can factor the updated diagonal block,

$$L_{22}U_{22} = A_{22},$$

using an un-blocked algorithm, and compute $L_{32} := A_{32}U_{22}^{-1}$. The algorithm is summarized in Algorithm 5.5. The memory reference pattern is illustrated in Figure 5.3.

ALGORITHM **5.5 Left-looking Block Variant**

```
* Left-looking block LU decomposition.

do i=1,t

     s=(i-1)*r +1          * Start position of block to decompose
     e=i*r                 * End position of block to decompose
     if i>1
          U_{1:s-1,s:e} := L^{-1}_{1:s-1,1:s-1} A_{1:s-1,s:e}      * STRSM
          A_{s:n,s:e} := A_{s:n,s:e} - L_{s:n,1:s-1} U_{1:s-1,s:e}  * SGEMM
     endif
```

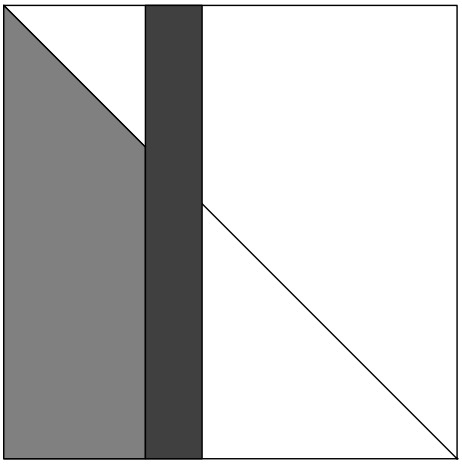


Figure 5.3: Left-looking block LU factorization. Lighter shade indicates elements that are used and darker shade indicates elements computed in the present stage.

Variant	BLAS	% operations	time	Mflops (average)
Left-looking	unblocked LU	10	20	146
	SGEMM	49	32	438
	STRSM	41	45	268
Right-looking	unblocked LU	10	19	151
	SGEMM	82	56	414
	STRSM	8	23	105

Table 5.2: Operations and times for block LU variants for $n = 500$, $r = 64$ on Cray 2-S, 1 processor.

$L_{s:n,s:e}U_{s:e,s:e} = A_{s:n,s:e}$ `* BLAS-2`

`enddo`

The performance of two variants of block LU decomposition is illustrated in Table 5.2 (data from [1]).

The main part of the work in block LU decomposition is based on the BLAS-3 routines, The variants differ in how much of the work is done by which subroutines. A subroutine may be more efficient or less efficient depending on a particular computer. It is therefore possible to optimize the algorithm for a specific architecture by choosing block size and variant of the algorithm.

5.3.1 LAPACK

LAPACK [2] is a library of subroutines for linear systems of equations, linear least squares problems, and eigenvalue problems. It is designed to replace both the LINPACK library which contains routines for solving linear systems of equations and least squares

and the EISPACK library for eigenvalue problems. LAPACK has been designed to give high efficiency on vector processors, high performance, workstations and shared memory parallel computers.

The subroutines are written in Fortran 77, and as much work as possible is performed by calls to BLAS routines, in particular block algorithms and BLAS 3 routines are used. This way the LAPACK programs are portable, and at the same time they perform well on most computers, in particular if the BLAS routines have been optimized for each computer. On some supercomputers (e.g. those from from Cray Research), the BLAS 3 routines are implemented with vector instructions and parallelization (if there are any idle processors at the beginning of the call to a BLAS routines then they are used for parallel execution of the code). In addition, the BLAS 3 routines are highly optimized, and the LAPACK routines parallelize automatically on Cray computers with more than one processor.

The linear equations part of the library contains routines for the solution of general linear systems, as well as banded, symmetric, positive definite, and indefinite systems. There are single and double precision routines for real and complex arithmetic.

The library contains routines for the computation of several matrix decompositions, e.g., LU, QR and SVD. Also a number of eigenvalue decomposition routines for symmetric and nonsymmetric matrices are included. Much effort has been made for providing comprehensive error bounds, both normwise and componentwise.

The matrix decomposition routines are block algorithms (see Chapter 5). To determine the block size the LAPACK routines call a subroutine, ILAENV, that returns the block size that is optimal for the actual computer, the LAPACK routine and the problem dimensions.

LAPACK was developed by an international group of researchers and it is available at no charge through *netlib* at URL

```
http://netlib.org/lapack
http://www.netlib.no/netlib/lapack/index.html
```

5.4 Message Passing LU Decomposition

We now study some implementation details of the right-looking (KJI) variant on a distributed memory parallel computer with message-passing. Consider the code

```
do k=1,n-1
   a(k+1:n,k)=a(k+1:n,k)/a(k,k)
   do j=k+1,n
      a(k+1:n,j)=a(k+1:n,j)-a(k,j)*a(k+1:n,k)
                              ! SAXPY operation
   enddo
enddo
```

Now, we implement this algorithm on a ring of p processes, and we will only consider neighbor-to-neighbor communication.

We first discuss two schemes for distributing the matrix to processes that will lead to inefficient usage of the parallel computer. Suppose first that we distribute the matrix over

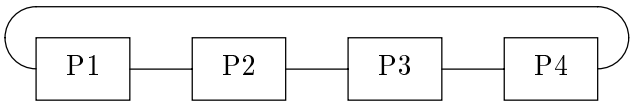


Figure 5.4: Ring of processes, $p = 4$.

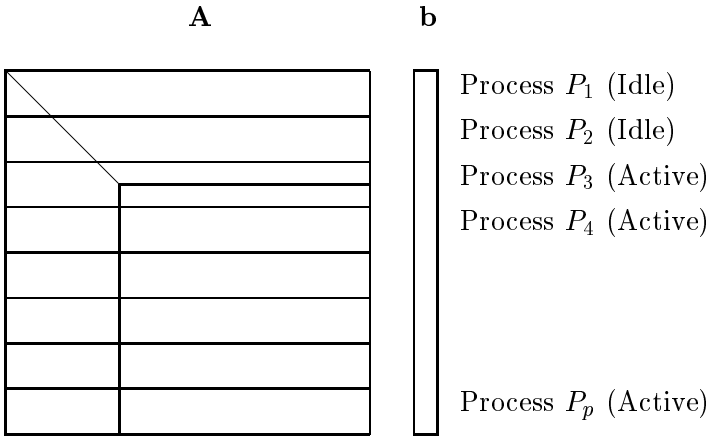


Figure 5.5: LU decomposition on a Block Row Partitioned Matrix.

the p processes in a block row fashion so that process P_i holds rows $(i-1)n/p+1$ to in/p of A and the corresponding components of the right hand side vector b, see Figure 5.5.

At step j, the row j which is stored in P_i must be sent to $P_{i+1} \ldots P_p$ in order to perform the eliminations in each of them.

Similarly, the matrix can be divided up into blocks of contiguous columns. Then, at step j, column j, which contains the multipliers is in P_i, and must be transmitted to $P_{i+1} \ldots P_p$. Since we are transforming A to upper triangular form, with these assignments, process 1 becomes idle after the initial n/p steps, then after n/p additional steps process 2 becomes idle, etc. Obviously, these assignments will not give a good load balancing.

On a sequential machine the time for LU decomposition is proportional to $\frac{2}{3}n^3\gamma$ where γ is the time required to perform 1 floating point operation. In the preceding schemes, use of p processes will not speed up the computation by a factor of p, no matter how fast the communication is, because processes are often idle. There are several ways of improving the efficiency of these algorithms. We could keep processes busy by having idle processes continue the elimination on rows above the pivot row instead of remaining inactive; this is the Gauss-Jordan method. An alternative is interleaving of rows or columns across processes or we distribute the columns to the processes in an cyclic way, as illustrated in Figure 5.6.

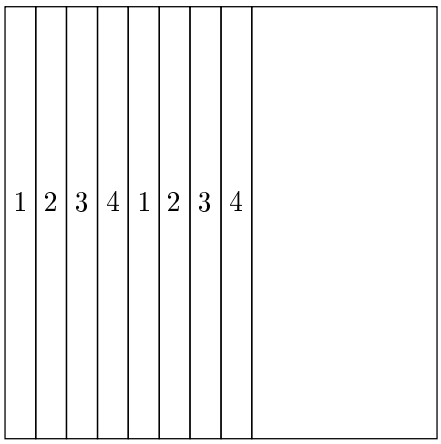


Figure 5.6: Cyclic distribution of columns to processes, $p = 4$. The numbers denote the process to which a column is assigned.

The pivot columns must be broadcast to all the processes, and this can be done in a pipelined fashion: As soon as a process receives the pivot column, it sends it to the neighbor to the east, and then starts to perform the updates to the local columns. The elimination is performed if each process executes the following code.

ALGORITHM **5.6 Message passing LU decomposition**

```
! LU decomposition, column cyclic distribution
! dimensions: n=r*p.
! columns of the local matrix C(1:n,1:r) are columns
!    A(:,myid), A(:,p+myid), A(:,2p+myid),...
!

j=0
do k=1,n-1
   if (mod(k,p)=myid) or (mod(k,p)=0 and myid=p) then
                                  ! Remainder when k is divided by p
                                  ! This process holds the
                                  ! pivot column
      j=j+1
      C(k+1:n,j)=(1/C(k,j))*C(k+1:n,j)
      send(east,C(k+1:n,j),p-1)
      piv_col(k+1:n)=C(k+1:n,j)
   else
      receive(west,piv_col(k+1:n),counter)
      if counter>1 then
         send(east,piv_col(k+1:n),counter-1)  ! Immediately
                                              ! send it further
```

```
      endif
   endif

   do jj=j+1,r
      C(k+1:n,jj)=C(k+1:n,jj)-C(k,jj)*piv_col(k+1:n)
   enddo
enddo
```

We use the variable `counter` to keep count of how many processes have had access to the present pivot column, and to prevent it from being sent around in the ring forever. When the program has been executed, each process holds `r` columns from the upper triangular matrix U in the LU decomposition of A. Under the diagonal in each column are the elements from the lower triangular factor L.

We will now perform a simplified timing analysis of Algorithm 5.6. Concerning the arithmetic, we disregard the computation of the pivot column, since this is one order of magnitude less work than the update of the rest of the matrix. Since we have a cyclic distribution of columns to processes, we have a good load balancing, and we can assume that the time for performing the floating point operations is $2n^3/(3p)\gamma$, approximately), where γ is the time for one flop.

Algorithm 5.6 is pipelined, and there is a startup phase, when the pipeline is filled, and a corresponding finishing phase, when it is emptied. If the problem is large, then these phases are much shorter than the time when all processes are busy[1], and therefore we will disregard them. In step k of the algorithm each process sends a message of length $n-k$, which, under the usual assumptions about the communication (Section 2.6.1), takes $\tau + \beta(n-k)$. Thus, the total communication time is

$$\sum_{k=1}^{n-1}(\tau + \beta(n-k))) \approx n\tau + \beta\frac{n^2}{2}.$$

The parallel efficiency of this algorithm is

$$E_{gec} \approx \frac{2n^3\gamma/3}{p(2n^3\gamma/(3p) + n\tau + n^2\beta/2)} = \frac{1}{1 + \frac{3p}{2n^2}\frac{\tau}{\gamma} + \frac{3p}{4n}\frac{\beta}{\gamma}}.$$

Thus, since the efficiency depends on p/n, this algorithm is *not scalable* (cf. Section 4.4.

In Sections 4.3.1 and 4.4 we saw that in order to get a scalable algorithm for matrix multiplication, it was necessary to distribute the matrices block-wise over a two-dimensional mesh of processes. If we use that distribution here, then we get a bad load balancing, as is illustrated in Figure 5.7. On the other hand, we have just seen that to obtain good load-balancing, it is necessary to use a cyclic distribution. As a consequence, to get a scalable algorithm for LU decomposition, with good load balancing, we should use a combination of the two approaches, where *blocks* of the matrix are distributed *cyclically* over the processes [12]. We illustrate this in Figure 5.8.

[1] In the startup and finishing phases there are altogether $2(p-1)$ messages sent, to be compared to n, approximately, in the middle part of the algorithm.

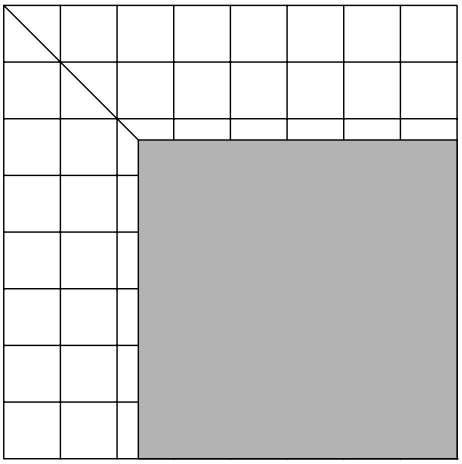


Figure 5.7: Bad load-balancing in the LU decomposition algorithm with blocked mapping. Each square represents a matrix block stored in one process. The processes completely outside the shaded area are idle.

We first derive the mathematical formulas for a right-looking block LU decomposition algorithm and then discuss the parallel implementation. Assume we have partitioned the matrix in $r \times r$ blocks, where $n = rm$. After $k-1$ steps of the block algorithm we have computed a partial triangularization of the matrix:

$$\begin{pmatrix} A_{11} & A_{12} & & & \cdots & & A_{1m} \\ & A_{22} & & & \cdots & & A_{2m} \\ & & \ddots & & & & \vdots \\ & & & A_{kk} & A_{k,k+1} & \cdots & A_{km} \\ & & & \vdots & \vdots & & \vdots \\ & & & A_{ik} & A_{i,k+1} & \cdots & A_{im} \\ & & & \vdots & \vdots & & \vdots \\ & & & A_{mk} & A_{m,k+1} & \cdots & A_{mm} \end{pmatrix}.$$

In the next step of the block elimination algorithm we zero the blocks in column k. This is equivalent to computing a block LU decomposition

$$\begin{pmatrix} A_{kk} & A_{k,k+1} & \cdots & A_{km} \\ \vdots & \vdots & & \vdots \\ A_{ik} & A_{i,k+1} & \cdots & A_{im} \\ \vdots & \vdots & & \vdots \\ A_{mk} & A_{m,k+1} & \cdots & A_{mm} \end{pmatrix}$$

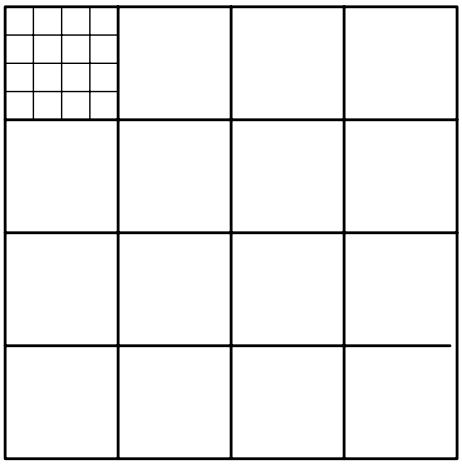


Figure 5.8: Block cyclic storage. The large squares represent large matrix blocks. Each large block is distributed over the whole array of processes. The small squares represent small blocks of the matrix.

$$= \begin{pmatrix} L_{kk} & & & & \\ \vdots & \ddots & & & \\ L_{ik} & \cdots & I & & \\ \vdots & & & \ddots & \\ L_{mk} & \cdots & & & I \end{pmatrix} \quad \begin{pmatrix} U_{kk} & U_{k,k+1} & \cdots & U_{km} \\ \vdots & \vdots & & \vdots \\ 0 & A'_{i,k+1} & \cdots & A'_{im} \\ \vdots & \vdots & & \vdots \\ 0 & A'_{m,k+1} & \cdots & A'_{mm} \end{pmatrix},$$

where the lower triangular matrix is zero except in the first block column and the diagonal. By identification we get

$$\begin{aligned} L_{kk}U_{kk} &= A_{kk}, && (5.5) \\ L_{ik}U_{kk} &= A_{ik}, \quad k+1 \le i \le m, && (5.6) \\ L_{kk}U_{kj} &= A_{kj}, \quad k+1 \le j \le m, && (5.7) \\ A'_{ij} &= A_{ij} - L_{ik}U_{kj}, \quad k+1 \le i,j \le m. && (5.8) \end{aligned}$$

First the blocks L_{kk} and U_{kk} are computed by "point" LU decomposition, then the elements in block row k and block column k by solving a sequence of triangular systems with multiple right hand sides. Finally, we compute the (i,j) blocks by matrix multiplication and addition. This block elimination algorithm is equivalent to the "point" algorithm (5.2), (5.3), except that here we also transform the elements of block row k.

In the discussion of the parallel algorithm it is instructive to consider Figure 5.9, which illustrates the situation just before step k has been performed. Since the diagonal "pivot block" is assigned to one process, that process can compute the LU decomposition (5.5) without communicating with any other process. Then the "pivot" process must broadcast U_{kk} to all other processes in the same process column (the process pivot column), and these processes will compute L_{ik} from (5.6). Similarly, L_{kk} is broadcast to all processes in the same process row (the process pivot row), and the blocks U_{kj} are computed. Finally,

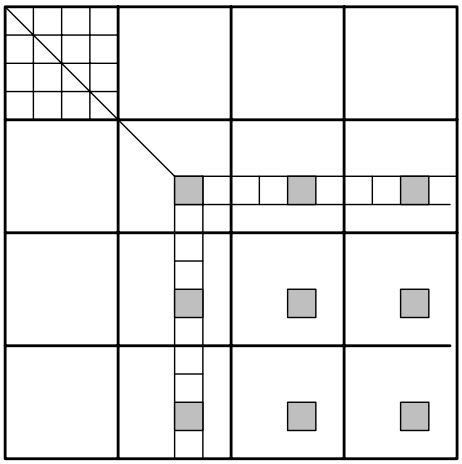


Figure 5.9: Block LU decomposition. We have $m = 4$, and the process grid is 4×4. In step k we will use the diagonal shaded block as "pivot block". The other shaded blocks are assigned to the same process as the diagonal block.

each process in column k broadcasts its block L_{ik} to all processes in the same row, and each process in row k broadcasts its block U_{kj} within its column, whereupon all the blocks A'_{ij} can be computed. Thus all communication is within process rows and columns.

ALGORITHM **5.7 Message Passing Right-Looking Block LU Decomposition**

```
do k=1,m
    if I hold pivot block then
        Compute LU decomposition L_kk U_kk = A_kk
        Broadcast U_kk within my process column
        Broadcast L_kk within my process row
    endif
    if I am in process pivot column then
        Receive U_kk
        Compute my blocks L_ik from L_ik U_kk = A_ik
        Broadcast L_ik within my process row
    endif
    if I am in process pivot row then
        Receive L_kk
        Compute my blocks U_kj from L_kk U_kj = A_kj
        Broadcast U_kj within my process column
    endif
    Receive L_ik and U_kj corresponding to all my blocks A_ij
```

```
    Perform updates A_ij := A_ij − L_ik U_kj
enddo
```

Of course, the broadcasts can be pipelined as in the column version. The detailed derivation of an estimate of the runtime for this algorithm is rather tedious [12]. It can be shown that the efficiency is

$$E_{lubc} = \frac{1}{1 + O(\frac{p}{n^2})\,\frac{\tau}{\gamma} + O(\frac{\sqrt{p}}{n})\,\frac{\beta}{\gamma}}.$$

Thus the algorithm is scalable.

The block cyclic distribution combined with the block algorithm has the advantages the pivot block is decomposed within one process only, the communication is always within process rows and columns, and each message is a whole block of numbers.

5.4.1 Pivoting

If we incorporate partial pivoting (which, in general, we must do, for stability reasons), then the algorithm becomes a little more complicated. We can no longer decompose the block A_{kk} separately from the rest of the block column, since in each step of the computation of L_{kk} we must search the columns of

$$\begin{pmatrix} A_{kk} \\ A_{k+1,k} \\ \vdots \\ A_{mk} \end{pmatrix},$$

for the element of largest modulus. The modified code is given below. Note that "pivot column" refers to the column of processes holding the matrix blocks A_{kk}, $A_{k+1,k}$, ..., A_{mk}.

ALGORITHM **5.8 Message Passing Right-Looking Block LU Decomposition with Partial Pivoting**

```
do k=1,m
    if I am in process pivot column then execute factor(A_kk, ..., A_mk)
    Receive pivoting information and apply interchanges
    if I am in process pivot row then
        Receive L_kk
        Compute my blocks U_kj from L_kk U_kj = A_kj
        Broadcast U_kj within my process column
    endif
    Receive L_ik and U_kj corresponding to all my blocks A_ij
```

```
    Perform updates A_ij := A_ij − L_ik U_kj
enddo
```

Before describing the procedure `factor`, we remark that the step where the rows are interchanged according to the pivoting information received applies also to blocks in L, and it involves exchanging segments of rows with other processes in the same process column.

The procedure `factor` computes the factorization

$$P_k \begin{pmatrix} A_{kk} \\ A_{k+1,k} \\ \vdots \\ A_{mk} \end{pmatrix} = \begin{pmatrix} L_{kk} \\ L_{k+1,k} \\ \vdots \\ L_{mk} \end{pmatrix} U_{kk},$$

where P_k is a permutation matrix, by Gaussian elimination with partial pivoting.

ALGORITHM **5.9** `Procedure factor`$(A_{kk}, \ldots, A_{mk})$

```
! Factor a block column of A with partial pivoting.
! All computations are performed within the blocks (A_kk, ..., A_mk)
do i=1,r
    Find maximum element and its location in column i
    Swap rows
    Scale column i
    Broadcast the pivot row to all processes in the process column
    Update the elements of the trailing submatrix
enddo
Broadcast pivoting information and all my blocks L_ik to processes within my
    process row.
```

Communication between processes in the process pivot column takes place in all steps in the loop of the procedure, except the update step.

The LU decomposition algorithm is scalable also with pivoting [12]. This is essentially the algorithm implemented in the ScaLAPACK library. The performance of the algorithm on an Intel Paragon computer is illustrated in Figure 5.10 (data adapted from [6]).

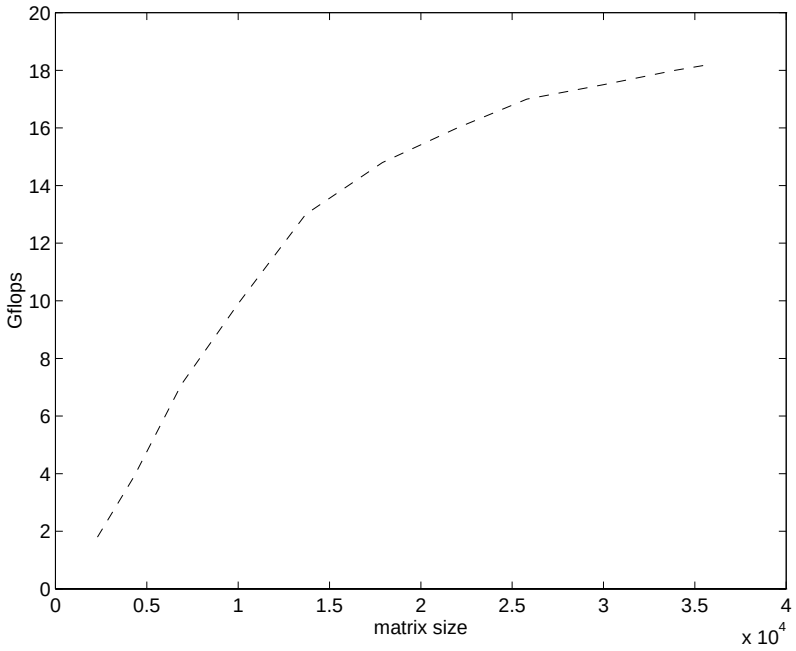


Figure 5.10: Performance of the ScaLAPACK LU decomposition algorithm on a 16×32 process grid on an Intel Paragon computer. The block size was $r = 8$.

5.5 Shared Memory LU Decomposition

As in the case of matrix multiplication, it is straightforward to parallelize Gaussian elimination. We use the column oriented SAXPY (KJI) version given on page 81, and instruct the compiler to parallelize the loop that performs the updates:

ALGORITHM **5.10 LU decomposition – OpenMP**

```
do k=1,n-1
   a(k+1:n,k)=(1/a(k,k))*a(k+1:n,k)
  !$omp parallel do schedule (static)
   do j=k+1, n
      a(k+1:n,j)=a(k+1:n,j)-a(k,j)*a(k+1:n,k)
   enddo
  !$omp end parallel do
enddo
```

Unless the matrix is extremely large, it is probably not worth it to parallelize the computation of the pivot column `a(k+1:n,k)` (the overhead for the fork-join is likely to be larger than the gain in parallelization). That computation is performed by the master thread.

The `do schedule (static)` means that the iterations of the loop are divided up in equal chunks, each of the size `(n-k)/p`, where `p` is the number of processors. By default, the loop index is made private to each processor. The computations in the parallel region

are completely independent, so there is no need to explicitly synchronize in this code. However, there is an implicit barrier at the `!$omp end parallel do` statement: each processor waits until the rest have finished their computations. This ensures that the pivot column for the next iteration has been been computed correctly, before it is transformed by the master thread.

5.6 Data Parallel LU Decomposition

To develop a data parallel program for LU decomposition, first consider the code

```
do k=1,n-1
   do i=k+1,n
      a(i,k)=a(i,k)/a(k,k)          % Divide by the pivot element
   enddo

   do j=k+1,n
      do i=k+1,n
         a(i,j)=a(i,j)-a(i,k)*a(k,j)
      enddo
   enddo
enddo
```

The main part of the work in the algorithm is in the `j,i`-loop. Assuming for the moment that each matrix element is stored in a separate processor, we consider the communication needed for modifying the (i,j) element in

```
a(i,j)=a(i,j)-a(i,k)*a(k,j)
```

In order for the (i,j) processor to modify its element, it must have `a(i,k)` and `a(k,j)`. The communication is illustrated in Figure 5.11.

This communication must be performed for all elements in the lower right submatrix, and it can be expressed using the Fortran 90 intrinsic function `spread`. The statement

```
spread(a(k,k+1:n),1,n-k)
```

creates a matrix of dimension `(n-k)`$\times$`(n-k)`, where the elements of each row are equal to the corresponding elements of row `k` in `a`. Similarly,

```
spread(a(k+1:n,k),2,n-k)
```

creates a matrix where the elements of each column are equal to the corresponding elements of column `k` in `a`. We illustrate this in Figure 5.12.

Now we can write the code for LU decomposition in the form

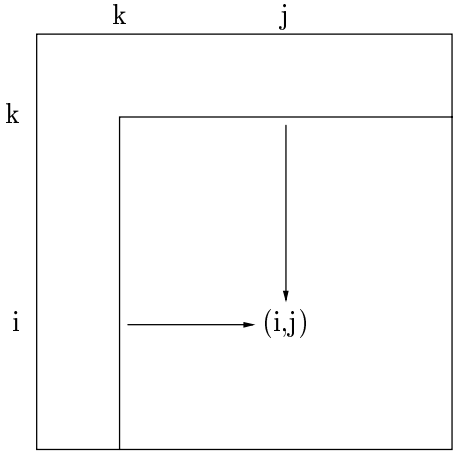


Figure 5.11: Communication for modifying `a(i,j)`.

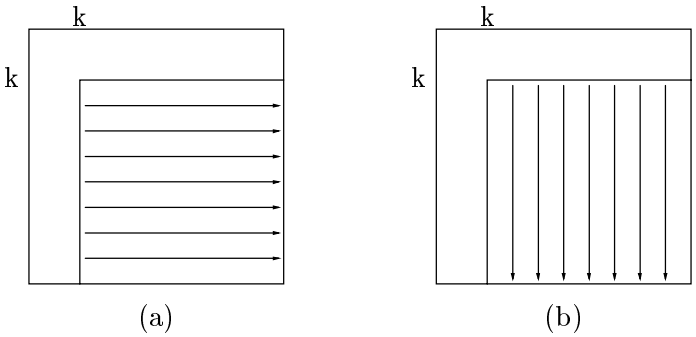


Figure 5.12: (a) `spread(a(k+1:n,k),2,n-k)` (b) `spread(a(k,k+1:n),1,n-k)`.

```
do k=1,n-1
   a(k+1:n,k)=a(k+1:n,k)/a(k,k)
   a(k+1:n,k+1:n)=a(k+1:n,k+1:n)-spread(a(k+1:n,k),2,n-k)*
                  spread(a(k,k+1:n),1,n-k)
enddo
```

The multiplication is element-by-element matrix multiplication, and each multiplication takes place in the processor where the element from `a` is stored.

Now assume that the matrix is too large to store one element per processor, and that it is assigned processors in a blocked fashion. Then after a few steps in the algorithm, some processors will be idle (Figure 5.7). As in the message passing algorithm we should use a block cyclic assignment as shown in Figure 5.8.

Bibliography

[1] E. Anderson and J. Dongarra. LAPACK working note 19, Evaluating block algorithms variants in LAPACK. Technical Report CS-90-103, Computer Science Department, University of Tennessee, April 1990.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, 1992.

[3] Babbage, H. P. Babbage's analytical engine. *Mon. Not . Roy. Astron. Soc.*, 70:517–26, 1910.

[4] Brainerd, Goldberg, Adams. Programmmer's Guide to Fortran 90. McGraw-Hill, 1990.

[5] Chandra et al, Parallel Programming in OpenMP, Morgan Kaufmann Publishers, 2000.

[6] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. Oak Ridge National Lab., Tech. Report ORNL/TM-12470, September 1994.

[7] Dongarra, J. J. Performance of Various Computers Using Standard Linear Equations Software. Computer Science Department, University of Tennessee, Tech. Rep. CS-89-85. See `http://www.netlib.org/benchmark/performance.ps`.

[8] Dongarra, J. J., Moler, C. B., Bunch, J. R., Stewart, G. W. LINPACK Users' Guide. SIAM, Philadelphia, 1979.

[9] J. J. Dongarra and D. W. Walker. Software Libraries for Linear Algebra Computations on High Performance Computers. SIAM Review 37(1995), 151-180.

[10] J. J. Dongarra, and R. C. Whaley. A User's Guide to the BLACS v1.0. LAPACK Working Note 94, University of Tennessee, Dept. Computer Science, June 1995.

[11] J.J. Dongarra, F.G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26:91–112, 1984.

[12] J. J. Dongarra, R. van de Geijn, and D. W. Walker. Scalability Issues Affecting the Design of a Dense Linear Algebra Library. *J. Parallel and Distributed Computing* 22:523-537, 1994.

[13] J. J. Dongarra, I. S. Duff, D. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers.* SIAM, Philadelphia, 1990.

[14] M. Flynn. Very High Speed Computing Systems. Proc. IEEEE 54(1966), 1901-1909.

[15] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comp.*, C-21:948–60, 1972.

[16] I. Foster. Designing and Building Parallel Programs. Addison-Wesley, Reading, Mass., 1995.

[17] Jalby W. Frailong, J. M and J. Lenfant. XOR schemes: A flexible data organization= in parallel memories. In *Proc. 1985 Internat. Conf. Parallel Processing*, pages 276–83. IEEE, 1985.

[18] D. Gannon and J. van Rosendale. On the impact of communication complexity in the design of parallel algorithms. *IEEE Trans. Comp.*, C-33(12):1180–1194, 1984.

[19] K. Gallivan, M. Heath, E. Ng, B.Peyton, R.Plemmons, J. Ortega adn C. Romine, A. Sameh, and R. Voight. *Parallel Algorithms for Matrix Computations.* SIAM, Philadelphia, 1990.

[20] R. A. van de Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm, Univ. Tennessee, Tech. Report CS-95-286, May 1995.

[21] G. H. Golub and C. F. Van Loan. *Matrix Computations. 2nd ed.* Johns Hopkins Press, Baltimore, MD., 1989.

[22] J. L. Hennessy and D. A. Patterson. *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann, San Mateo, 1990.

[23] High Performance Fortran Forum. High performance Fortran language specification. Technical report, Computer Science Department, Rice University, 1993.

[24] R. W. Hockney and C. R. Jesshope. *Parallel Computers.* Adam Hilger Ltd, Bristol, 1981.

[25] R. Hockney. Performance of Parallel Computers. In Proceedings of the NATO Workshop on High Speed Computations, ed. J. Kowalik, NATO ASI Series, vol. F-7, p. 159-176, 1984.

[26] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, New York, 1993.

[27] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, and M. E. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, Mass., 1994.

[28] V. Kumar, A. Grama, A. Gupta, and G. Karypis. Introduction to Parallel Computing. Benjamin/Cummings, Redwood City, Ca, 1994.

[29] J. Levesque and J. Williamson. *A Guidebook to Fortran for Supercomputers.* Academic Press, San Diego, CA, 1989.

[30] C. A. Mead and L. A. Conway. *Introduction to VLSI Systems.* Addison Wesley, Mass. ,USA, 1980.

[31] Message Passing Interface Forum. Draft document for a standard message passing interface. Technical report, Computer Science Department, University of Tennessee, November 1993.

[32] Metcalf, M., Reid, J. Fortran 90 Explained, Oxford University Press, Oxford, 1990.

[33] A. H. Sameh. Numerical parallel algorithms - a survey. In A. Sameh D. Lawrie, editor, *High Speed Computer and Algorithm Organization=*, pages 207–228, New York, 1977. Academic Press.

[34] R. Schreiber. High Performance Fortran, Version 2. Parallel Processing Letters 7(1997), 437-449.

[35] C. L. Seitz. The cosmic cube. *CACM*, 28:22–33, 1985.

[36] H. D. Shapiro. Theoretical limitations on the efficient use of parallel memories. *IEEE Trans. Comp.*, C-27:421–28, 1978.

Index