수 학 강 의 록

제 15 권

Numerical Linear Algebra Algorithms on Vector and Parallel Computers

LARS ELDEN AND HAESUN PARK

서 울 대 학 교
수 학 연 구 소 · 대 역 해 석 학 연 구 센 터

Notes of the Series of Lecture
held at the Seoul National University

Lars Elden, Linköping University, S-581 83 Linköping, Sweden
Haesun Park, University of Minnesota, Minneapolis, MN 55455, U.S.A.

퍼낸날 : 1993년 8월 14일
지은이 : Lars Eldén and Haesun Park
퍼낸곳 : 서울대학교 수학연구소 · 대역해석학연구센터〔TEL : 02-880-6530〕

Contents

Chapter 1

Introduction

1.1 What is a Supercomputer?

There is no formal definition of a **supercomputer**, but a common way of explaining the concept is to say that a supercomputer is a computer of which the *peak performance is at least 10% of the presently fastest existing computer.* This makes the definition a loose one, which changes over time, and this is reasonable since the development of supercomputers is very fast: The performance of a powerful, expensive and big supercomputer some ten years ago can now be found in a relatively cheap desktop computer that is available for use by a single researcher or engineer.

A similar, loose definition is the following: a supercomputer is a very powerful computer, at least two orders of magnitudes more powerful in terms of speed and storage than a conventional computer.

To illustrate the aspect of speed, we cite the following performance figures from a commonly used computer benchmark, namely the LINPACK benchmark [5]. In that test the performance of computers is measured from the execution times when two linear systems of equations are solved: the first is of dimension 100 and is solved using a subroutine from the LINPACK library[3]. The second is larger (dimension 1000) and here any program (i.e., even heavily optimized) can be used. Below we give timings in Mflops (million floating point operations per second) for some supercomputers and workstations.

This shows that for the fastest supercomputers and workstations (April 1992) and for problems that are worth solving on supercomputers (i.e., problems that are large in some sense, usually in terms of the number of floating

Computer	$n = 100$	$n = 1000$	Peak performance
Cray Y-MP C90 (16 proc.)	479	9715	15238
NEC SX-3/44R (4 proc.)		15120	25600
Fujitsu VP2600/10 (1 proc.)	249	4009	5000
IBM Risc/6000-560	31	84	100
SUN SPARCstation IPX	4.1		

Table 1.1: LINPACK Benchmark for various computers, May 31, 1993.

points operations that are needed) it is really true that the supercomputers are two orders of magnitude faster.

Since there is no real definition of the term "supercomputer", and since it gives the impression that a particular computer is something special, it can be used by computer manufacturers as a marketing trick. Therefore, many prefer the term "high performance computer", which can be interpreted in a wider sense, and is less prestigious. We will use the term supercomputer in the sequel.

Supercomputers are made fast by introducing parallelism on different levels in the architecture. We will refer to non-supercomputers as *conventional* or **sequential computers**, although such computers normally have some parallel features (especially on a low level of the hardware), but to a lesser extent than supercomputers. The first computer, which was labeled supercomputer, was the Cray-1. This machine appeared in 1976.

1.2 Why Supercomputers?

Supercomputers are designed to be very fast and are intended for problems that would otherwise be intractable. Here 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 used as measures of 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 at a speed of 60 km/h. 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 much in a product development stage, when it is necessary to evaluate several alternative constructions. On the other hand, with a supercomputer that can run at 1 Gflops, the same computation takes about 35 minutes, which is acceptable.

The second example is concerned with the flow around the Hermes 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 grid point 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.

In reducing computation time in a rate that is comparable to the improvement seen since 1950's, it is necessary to use parallelism. From 1950 to the mid 1970's, the improvement of an order of 10^5 was made in speed, which was due to improvements in the clock cycle and in architecture and design. Since gains from the improvement in semiconductor technology are becoming much harder, parallelism is the only means for making orders of magnitude improvement in computing speed. The hardware to deliver such performances have been built but they are not easy to exploit efficiently. Therefore, the users have to understand the architecture and be able to redesign their algorithms to benefit fully from new technology.

Chapter 2

Architecture

2.1 Basic concepts

We shall here briefly discuss same basic concepts that are needed in the sequel for describing supercomputers and for discussing their performance. Since the emphasis of this text is the *use of supercomputers* for linear algebra problems rather than the *construction of supercomputers*, we will not go into details concerning hardware.

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 even increase it by a larger factor) by introducing more parallelism.

It is interesting to note that the cycle times of supercomputers has not been decreased very much, since the first supercomputer, the Cray-1, was in-

troduced in 1976. The Cray-1 had a cycle time of 12.5 ns. The presently fastest supercomputers have cycle times of the order of a couple of nanoseconds. However, present day machines are at least a factor 100 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.

In principle, a signal that is sent from one part of the computer should be able to reach any other part during one clock cycle. Therefore, the speed of light is one factor that limits performance. The computer must be so small that signals can reach their destination during one cycle. To give an idea of the distances involved, we note that the distance traveled by light in 1 ns is about 30 cm.

To illustrate **scalar computations** we take the simplest possible example, the addition of to real numbers,

```
s1=s2+s3
```

The machine operations needed to execute this statement are (we use an informal assembler type notation here and in the sequel; the number of cycles given are only meant to be an example).

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

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

Sequential computers do everything in **scalar mode**. Thus 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.

In the next section we will discuss how the time for this loop can be reduced by parallelism.

Later we will deal extensively with vectors: 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
```

A **word** is the basic unit of information that is addressed in a computer.

The IEEE standard for binary floating point arithmetic was adopted in 1985. It defines four floating point formats in two groups, **basic** and **extended**, each with two widths, **single** and **double**. The organization of the double precision basic format is shown in Figure 2.1.

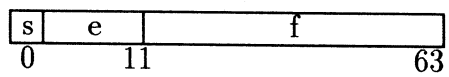


Figure 2.1: Basic format, IEEE double precision.

The components are the sign s (one bit), the exponent e (eleven bits), and the fraction f (52 bits), altogether 64 bits. The value v of a floating point number x is

$$v = (-1)^s(1.f)2^{e-1023} \qquad \text{for } 0 < e < 2047,$$

We see that the leading bit of the significant, i.e., the one to the left of the binary point, is not stored, since due to the normalization it is known always to be equal to one. Thus, one extra bit is gained for the fraction. The largest and smallest positive numbers that can be represented are $2^{1023}(2-2^{-52}) \approx 9 \cdot 10^{307}$ and $2^{-1022} \approx 2 \cdot 10^{-308}$, respectively. Implementations of the standard provide the addition, subtraction, multiplication, division and square root operations, as well as binary-decimal conversions. Except for the conversions, all operations give a result that is equal to the rounded result of the corresponding operation correct to infinite precision.

Most modern computers implement the IEEE standard (the Cray Research supercomputers are a notable exception: their word length is 64 bits, but they have a different floating point format), and since most supercomputers have word length 64, their normal floating point format is double precision.

As a first example of parallel computations we consider the problem of computing an inner product of two vectors $x \in R^n$ and $y \in R^n$,

$$s = \Sigma_1^n x_i y_i \tag{2.1}$$

A typical algorithm would look like

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

Although indices are referenced in the order of $1, 2, \cdots, n$, there are numerous other ways of arranging them. If p processors are available for the computation and $m = n/p$, we can rearrange the above as

```
do j = 1, p in parallel
 s(j) = 0
 do i = (j-1)*m+1, j*m
   s(j) = s(j) + x(i)*y(i)
 enddo
enddo
s = 0
do j = 1, p
  s = s + s(j)
enddo
```

When there are more processors, we can apply the above idea recursively and design a better parallel algorithm.

A dependency graph is a graph that represents the dependence among tasks in an algorithm. The nodes in the graphs represent the tasks in the algorithm and the directed edge from node i to node j represents that task j can start only after task i is completed. The following three figures show the dependency graph of the above inner product algorithms. The graphs allow us to identify the tasks that can be executed in parallel. The height of the graph represents the minimum number of sequential steps in the algorithm.

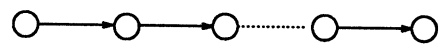


Figure 2.2: Dependency graph 1

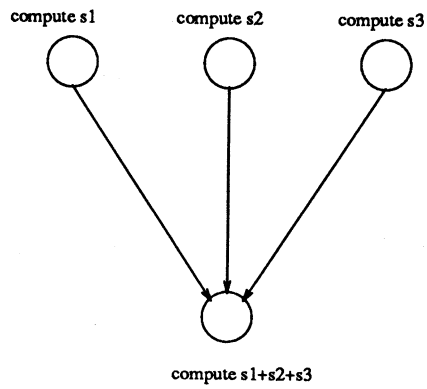


Figure 2.3: Dependency graph 2

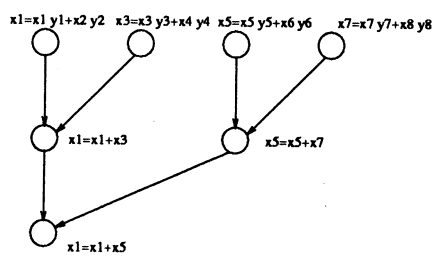


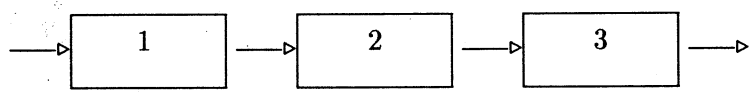
Figure 2.4: Dependency graph 3

2.2 Forms of Parallelism

2.2.1 Pipelining

We will use the term **functional unit** for the hardware that performs different arithmetic and logical operations in the supercomputer. We are mainly interested in the floating point addition, multiplication and division operations. In this section, we discuss how these operations are efficiently executed on many modern computers using pipelined functional units. This concept is described with addition as an example.

To introduce the concept of a pipeline, 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).

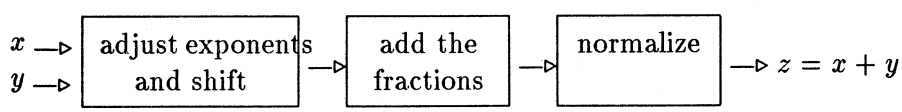


The normal operation of such an assembly line is to input enough material for one car into the procedure every time unit, so that the workers are active all the time and produce one car every time unit.

Next, consider floating point addition. The following example shows that floating point addition can be divided into stages:

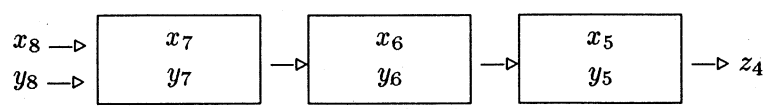
$$1.234 \cdot 10^0 + 4.567 \cdot 10^{-2} = (1.234 + 0.04567) \cdot 10^0 = 1.27967 \cdot 10^0 \doteq 1.280 \cdot 10^0.$$

For simplicity, we assume here three 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, a **pipeline**: 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 2.5

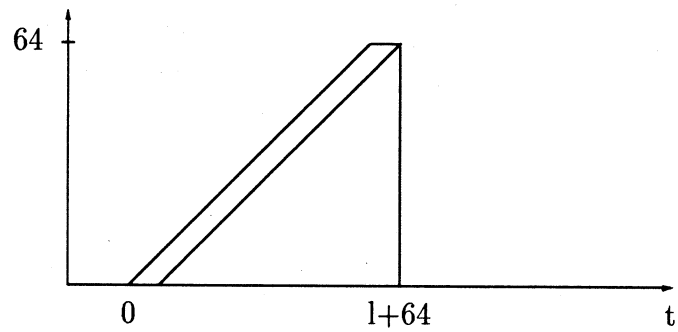


Figure 2.5: Timing diagram for a pipelined operation with $n = 64$. The pipe length is assumed to be l.

The computation in **vector mode** of the sum of the two vectors takes $l + n$ clock periods, where l is the length (in cycles) of the pipeline and n is the vector length. The corresponding computation in **scalar mode** (without pipelining) would take ln clock periods.

2.2.2 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 supercomputers with pipelined arithmetic have **vector registers**, which can be considered as intermediate storage between the functional units and the primary memory.

In the Cray X-MP (and Y-MP) processor, there are 8 vector registers, each with 64 elements. For our examples with vector registers we will assume that they are of size 64.

The vector registers are used in **vector instructions**. As an example, consider the following code:

```
do i=1,64
```

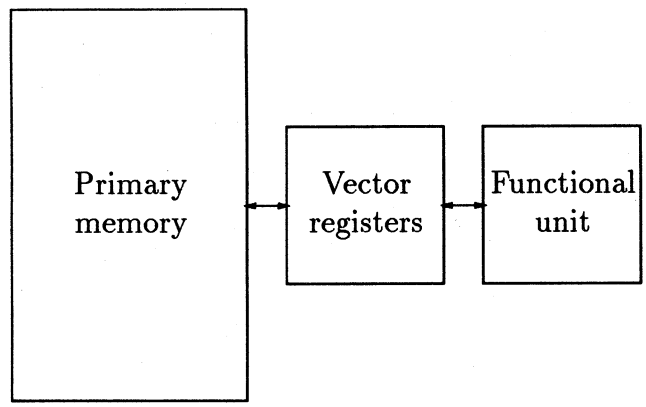


Figure 2.6: Vector registers are intermediate, fast storage units between primary memory and functional units.

```
   a(i)=b(i)+c(i)
enddo
```

On computers with vector instructions the (informal) assembler code for this is:

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

Thus, there are only four machine instructions. We also give a more detailed assembler version of the same code:

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

Here 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 (operations). A compiler that can take a program written in a high level language and produce code with vector instructions is called a **vectorizing compiler.**

Note that a vector operation can not be stopped once it has started: 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 number of operations in a vector operation is larger than the length of the vector registers, then the loop must be divided up in sub-loops. Consider, e.g.,

```
do i=1,n
   x(i)=y(i)*z(i)
enddo
```

where $n > 64$. The compiler must generate 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.
do i=1,rem
   x(i)=y(i)+z(i)
enddo
i=rem
do j=rem+1,n,64
   do k=1,64
```

```
            i=i+1
            x(i)=y(i)+z(i)
         enddo
      enddo
```

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

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

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

where t_c is the clock period. The rate of producing n results is

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

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

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

which gives

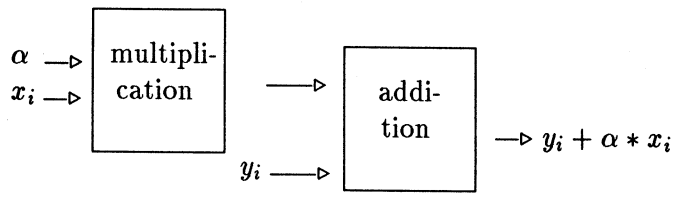
$$n_{1/2} = l.$$

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

2.2.3 Chaining

Other vector operations that can be pipelined are (componentwise) multiplication $z_i := x_i * y_i$, $i = 1, 2, \ldots, n$, and division $z_i := y_i/x_i$, $i = 1, 2, \ldots, n$. It

is also common that two arithmetic functional units can be **chained** together to form a single pipeline. By chaining the multiplication and addition units,



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

```
do i=1,64
   y(i)=y(i)+s*x(i)
enddo
```

where s is a scalar, can be performed in vector mode, i.e., so that one result is produced every clock period. Similarly, the code

```
do i=1,64
   y(i)=y(i)+z(i)*x(i)
enddo
```

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

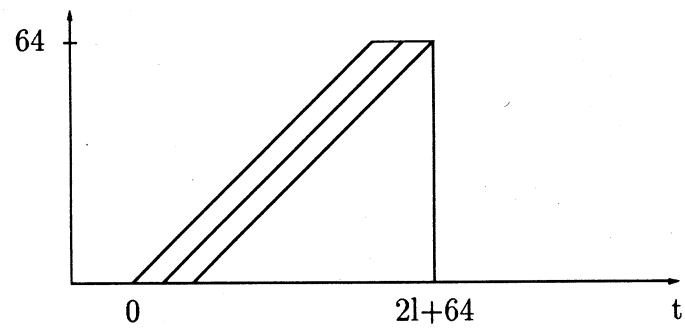


Figure 2.7: Timing diagram for two chained vector operations.

It is seen that 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 has 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.

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 [11]).

2.2.4 Indirect Addressing

In sparse matrices, the majority of the elements are equal to zero. It is common in applications that such matrices are of the order $10^5 - 10^6$, but the number of nonzero elements is less than 5%. For 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 are 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`:

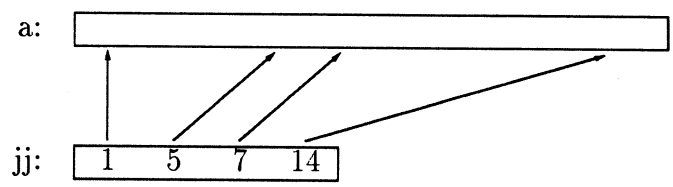


Figure 2.8: The vector `jj` with addresses elements in `a`.

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
```

2.2.5 Conditional Statements

As we said earlier, vector instructions cannot be interrupted. This means that in order to vectorize conditional statements, special arrangements are needed. The **vector mask (VM) register** is a register with 64 positions, each one bit wide. It can be used to vectorize the following type of statements.

```
do i=1,n
   if (a(i) > 0) x(i)=y(i)*a(i)
enddo
```

The following assembler code illustrates how the VM register is used.

```
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
```

All instructions are vector operations. On some computers (but not on Cray's) arbitrary vector operations can be controlled by the VM register.

There are two problems with this construct:

- If only a few of the conditions are true, then many arithmetic operations are wasted. It may be much faster to execute the loop using scalar instructions. Note that the compiler cannot decide this in advance.

- It can not be used for codes of the type

```
do i=1,n
   if (a(i) > 0) x(i)=y(i)/a(i)
enddo
```

since that would lead to division by zero.

2.3 Memory Organization

We start this section with a quotation from one of the pioneers in the early history of the digital computer.

> *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)

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

Typically, the **memory cycle time** (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 it is four cycles. It is obvious that with such a memory speed, the vector load (and store) operations cannot deliver operands to (and from) vector registers with a speed that matches that of the vector floating operations (this is needed if memory accesses and floating point operations are to be chained, as shown earlier).

In order to increase the performance of the memory, it is divided into separate units, which can operate in parallel. Such memory organization is called **interleaved**, see the next section.

As memory access patterns are important on high performance computers, we need to specify here how matrices are stored in primary memory. In Fortran, which is the most commonly used language for scientific computations, matrices are stored in column major order. E.g., a 3×3 matrix A is stored

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

There are programming languages where matrices are stored row-wise, e.g., C and Pascal. In the following, we assume the Fortran storage convention.

2.3.1 Interleaved Memory

The primary memory of many high performance computers is interleaved: it consists of separate banks, which can operate independently of each other. Reading or writing a word from a bank takes a certain number of clock cycles.

The consecutive elements of a vector are stored in consecutive banks, see Figure 2.9

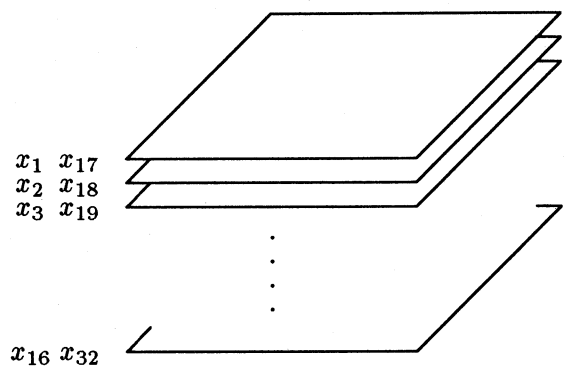


Figure 2.9: The storage of a vector in interleaved memory with 16 banks.

As an example, on the Cray X-MP/416 the primary memory is interleaved with 64 banks.

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 the 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: 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. Note that since matrices are stored in column major order, referencing a matrix row-wise is equivalent to referencing a vector with non-unit stride.

With interleaved memory also `vload` (vector load) and `vstore` (vector store) operations can be chained. The execution of the assembler code

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

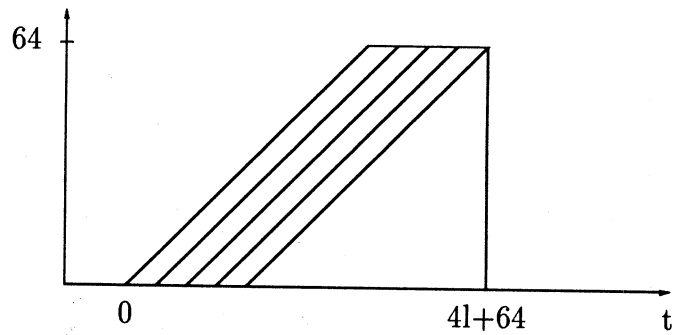


Figure 2.10: Timing diagram for four chained vector operations.

can then be illustrated by the timing diagram in Figure 2.10.

If memory bank conflicts occur during the execution of a vector load or store operation, then the floating point operations are delayed (but not interrupted).

In order to perform the above code as a chained vector operation 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, in a computer with multiple processors, two different processors access the same bank, and again that conflict is resolved by special hardware. Note that interleaved memory with many banks is particularly useful in computers, where several processors share the same memory, since this reduces the risk of this type of memory conflict.

We finish this section with a diagram (Figure 2.11) showing the basic structure of a vector register architecture.

2.3.2 Memory Hierarchy

Another important concept in high performance computers is **memory hierarchy**. Due to the cost of manufacturing very fast memory hardware, computer designers often must compromise between memory speed and size. Many modern high performance computers, therefore, have a memory hierarchy, see Figure 2.12. (We remark that some computers have vector registers and no cache memory, while others have cache and no vector registers. There are computers that have both.)

Obviously, it is desirable to *perform as many floating point operations*

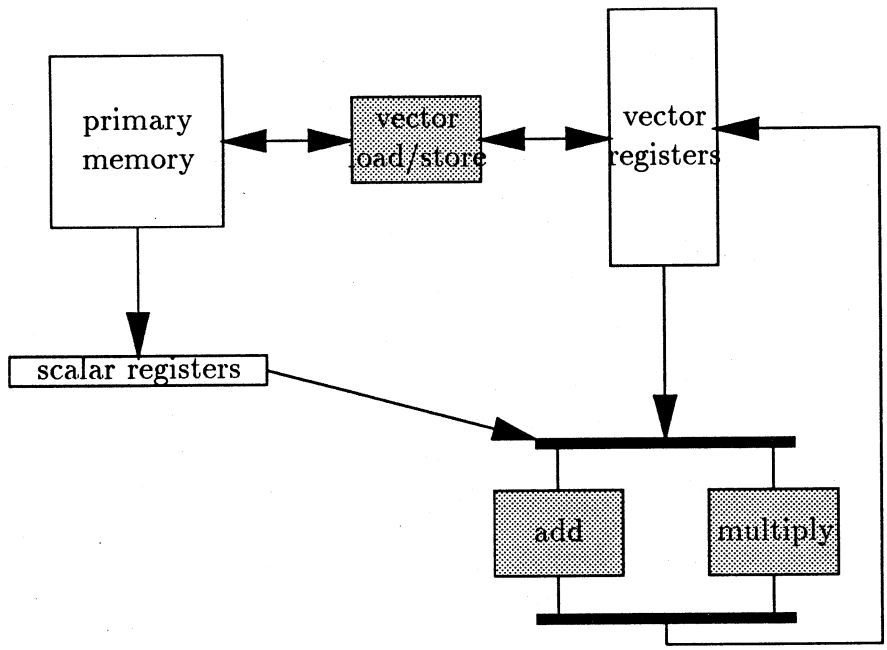


Figure 2.11: Vector register architecture. Pipelined units are shaded.

as possible for every floating point variable that is transferred from primary memory to the registers or cache memory.

2.4 Shared Memory Parallel Computers

Multiprocessor architectures with shared memory are tightly coupled systems in which there is complete connectivity between processors and memory modules. A simplified block diagram is shown in Figure 2.13.

The primary memory may be centralized (only one memory module) or partitioned into several modules. The common memory is accessed by all processors. The interconnection network is a potential bottleneck for these systems. Memory contention (memory access conflict) is important 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

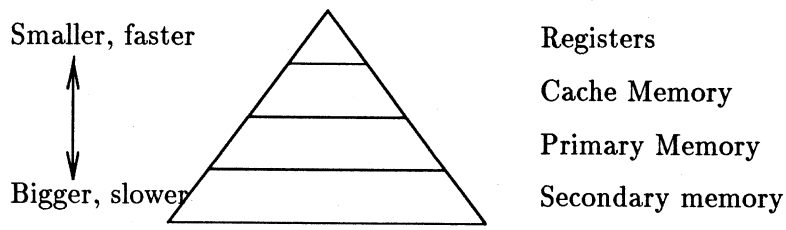


Figure 2.12: Memory hierarchy

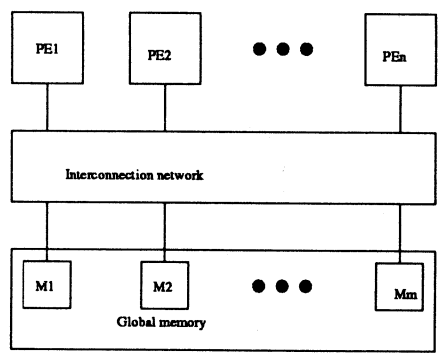


Figure 2.13: Shared memory systems

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. Examples of shared-memory systems include the Alliant, the Encore Sequent, the Cray-1, Cray-2, Cray Y-MP, Fujitsu VP2600, NEC SX-3.

2.5 Distributed Memory Parallel 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 several computer modules and an interconnection network, see Figure 2.14.

Each computer module has a processor, a memory, and an I/O interface. Data communication is carried out through message passing. Each message usually consists of a number of fixed-size packets. Inter processor communication follows a predetermined communication protocol.

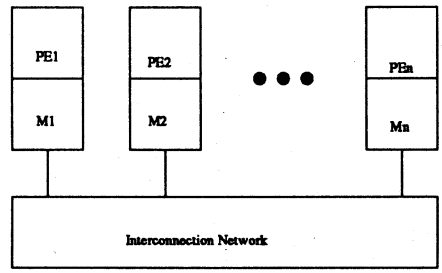


Figure 2.14: Distributed memory system

2.5.1 Interconnection Networks

We present several interconnection topologies that are commonly used in message passing system.

Linear and Ring Arrays In a linear array, p processors are connected along a line and in a ring array, they are connected around a ring as shown in Figures 2.15 and 2.16.

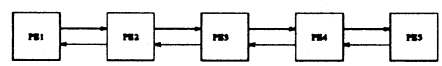


Figure 2.15: Linear array of processors

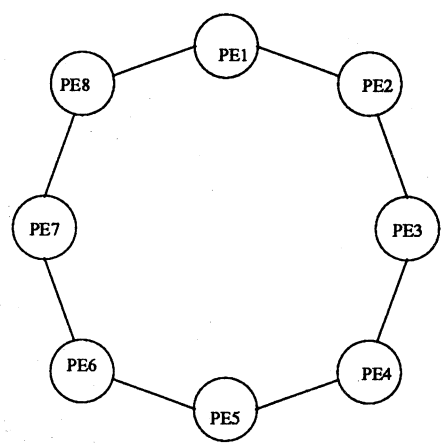


Figure 2.16: A processor ring

Two-dimensional arrays A two-dimensional array consists of an array of

$p \times p$ processors connected as shown in Figure 2.17. Often, the wrap around connection is assumed in order to yield more homogeneous complexity results.

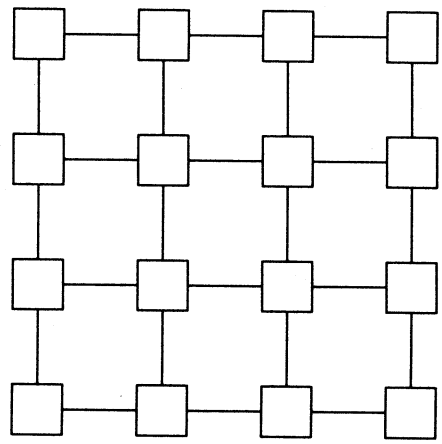


Figure 2.17: A two-dimensional array

Hypercubes Hypercubes are based on the binary n-cube topology and they are called by several different names such as Cosmic cube, n-cube, binary n-cube, etc. The first hypercube system was built at Caltech in the early 1980s as an experimental parallel computer for scientific computations.

A hypercube multiprocessor consists of 2^n processors, consecutively numbered with binary integers using a string of n bits. Each processor is connected to every other processor whose binary number differs from its own by exactly one bit. Hypercube interconnection networks for $n = 1, 2, 3, 4$ are shown in Figure 2.18 A hypercube of order 0 has one node, and the hypercube of order $n+1$ is constructed by taking two hypercubes of order n and connecting their respective nodes.

The hypercube interconnection network has several important properties such as the fact that the number of connection wires increases only logarithmically as the number of processors increases. Furthermore, several other interconnection networks such as linear and ring arrays, two-dimensional arrays, and trees, can be embedded efficiently in the hypercube, which means that the embedding can be done in a way so that that neighbors in the embedded graph become neighbors in the host graph (the hypercube).

The path length for a message between any two nodes is exactly the number of bits in which their tag bits differ. The maximum is n and numerous possible paths connecting any two nodes exist.

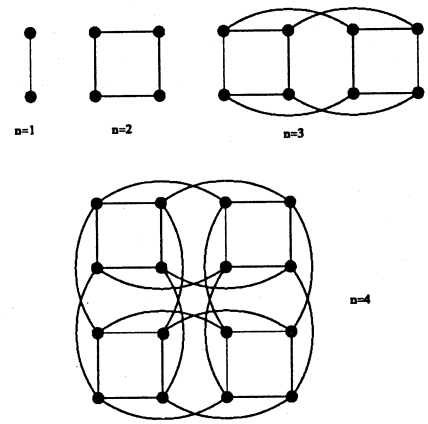


Figure 2.18: Hypercube interconnection

Fat Trees Consider a binary tree and locate computation processors at the leaves (squares in Figure 2.19). Place communication control processors at the other nodes (filled circles).

Real trees become thicker further from the leaves. Fat trees resemble real trees in the sense that the bandwidth increases further from the leaves. One common scheme is to double the communication capacity (bandwidth) for every level as we ascend from the leaves to the root.

Assume that the wires are full duplex at the leaf level. This means that any two neighboring leaves, e.g. P0 and P1 in Figure 2.19, can send and receive messages from each other at the same time. Since the bandwidth is doubled on the next level, there is enough capacity to let P0 and P2 communicate with each other at the same time as P1 and P3 communicate. Similarly, it is easy to see that the bandwidth is large enough for the situation when each processor is communicating with one other processor.

The Connection Machine CM-5 has a network based on the fat tree.

2.6 SIMD and MIMD Parallel Computers

Processors in various parallel processing architectures operate either under the centralized control or work independently. In SIMD (Single Instruction and Multiple Data streams) architectures, there is a single control processor which dispatches instructions to each processor. The same instruction is executed by all processors. Processors in MIMD (Multiple Instruction and Multiple Data

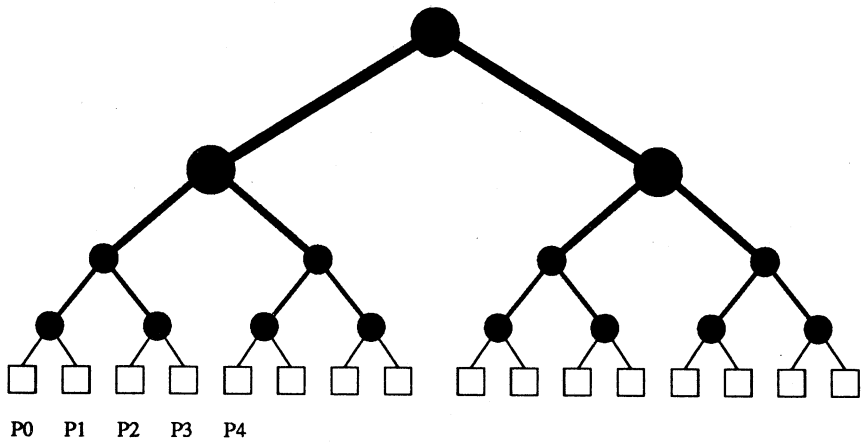


Figure 2.19: Binary fat tree. The processors are located at the leaves. The circles represent communication control processors, and the arcs are the communication channels.

streams) architectures do not have this kind of external control and they can execute different programs asynchronously. Any synchronization results from a possible need for exchanging data with other processors.

In 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 SIMD computers, all the processors should execute same instruction in the same clock cycle. This means that each condition should be executed serially in a conditional statement. Many unstructured applications are better suited to MIMD computer. SIMD computer offer free synchronization after each instruction execution and it is better for parallel programs that require frequent synchronization.

2.7 Performance Measurements

2.7.1 Speedup and Efficiency

In ideal situation, one expects to gain a factor of p in time when using p processors to solve a given problem. If T_1 and T_p are the time for executing the algorithm on one processor and p processors, respectively, then the speedup is

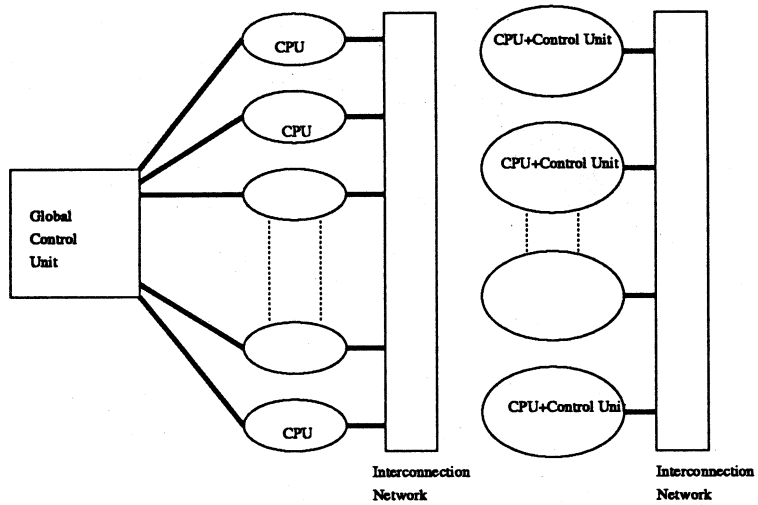


Figure 2.20: Typical SIMD and MIMD architectures

$$S_p \equiv T_1/T_p.$$

Often the parallel algorithms are not the best ones on one processor and this can lead to the following alternative which compares the best possible algorithm on one processor to the parallel algorithm on p processors

$$S_p^a \equiv T_s/T_p,$$

where T_s denotes the time required for the best serial algorithm that solves the problem.

The efficiency is the speed-up divided by the number of processors,

$$E_p = S_p/p \text{ or } E_p^a = S_p^a/p.$$

2.7.2 Amdahl's Law

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. Given a program, does it pay to run it on a supercomputer (in the sense that it utilizes the hardware efficiently)?

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

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

Further assume that the fraction f_v of the total number of operations in the code can be 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

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

In Figure 2.21 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 (the fraction of the code that is vectorizable) 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 2.21, we then get a performance of only 36% of peak performance. Thus, Amdahl's law give a rather pessimistic picture of the usefulness of supercomputers. However, it is a little misleading in many situations.

When one is evaluating a new supercomputer, it is very likely one has in mind running 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. One can think of Gaussian elimination with partial pivoting, where the transformations of the matrix elements require $O(n^3)$ operations

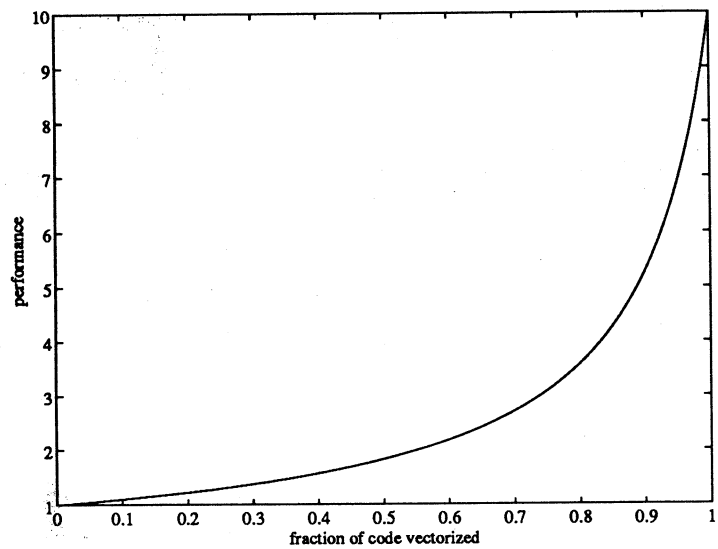


Figure 2.21: Amdahl's law. We have used $t_s = 10t_v$, and scaled so that peak performance r_∞ is equal to 10.

and can be vectorized well. The pivot search, on the other hand, takes $O(n^2)$ operations and vectorizes badly.

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 following 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 2.21 we now get 82% of peak performance.

On a parallel computer, let us assume that the fraction of serial part in an algorithm is f_1. If the total execution time on one processor is t_1, the execution time for the intrinsically sequential part is $f_1 t_1$ and the rest is $(1 - f_1)t_1$. On

p processors, the sequential part cannot be parallelized, so its execution time will be the same and the parallel part can be parallelized by a factor of p, which gives

$$t_p = f_1 t_1 + (1 - f_1)t_1/p.$$

Thus, the speed up is bounded by

$$s_p \leq 1/(f_1 + (1 - f_1)/p))$$

which means that for any algorithm and any number of processors,

$$s_p \leq 1/f_1$$

and this indicates that the efficiency will decrease asymptotically to $1/(pf_1)$.

2.7.3 Scaled Speed-up

According to Amdahl's law for a parallel computer, the only way to increase efficiency is to reduce f_1. Often, a way to reduce f_1 is to increase the problem size. Note that the parallel overhead will have more effect when the problem size is smaller. This observation lead to the definition of *scaled speed-up.*

In Amdahl's law, by assuming that the fraction of the sequential part is a constant, it assumes a given problem size which takes a certain fixed time on a sequential machine. If we assume that what is fixed is the time on a p-processor machine, t_p^s is the time to execute a given program on a p-processor machine, and f_1^s is the fraction of the sequential part, then the time for the same program run on a single processor would be

$$t_1^s = f_1^s t_p^s + p(1 - f_1^s)t_p^s$$

and the speed up is

$$s_p^s = (f_1^s t_p^s + p(1 - f_1^s)t_p^s)/t_p^s = f_1^s + (1 - f_1^s)p.$$

In the scaled speed up, we assume that the time to solve the given problem on a p-processor computer is fixed, which means that the problem size increases as the number of processors increase.

We can further modify the above speed up and calculate the speed-up by allowing the problem that is as large as can be fit in memory. Define

$$s_p^G = w_p T_1/T_p$$

where T_i is the time for executing Q_i, Q_i is the maximum size problem that can be solved on an i-processor computer, and w_p is the adjustment factor, which accounts for the difference in the number of arithmetic operations due to the difference in problem sizes. The number of operations for solving Q_p is w_p times the number of operations required to solve Q_1. The adjustment by w_p guarantees that we are comparing two executions that will perform the same number of operations.

2.8 Important Issues in Parallel Computers

Identification of Parallelism The parallelism can be expressed either by users or compilers. Automatic parallelization of sequential programs for multiprocessors has been only partially successful.
Partitioning After parallelism is identified, we need to partition the computational task into processes and identify the objects that they share.
Memory Allocation The creation of new processes requires allocation of memory space. Memory allocation is influenced by the memory organization and by the interconnection network.
Memory Access In shared-memory multiprocessors, this is particularly important since processors compete for same memory locations. A problem in multiprocessors is that of maintaining memory consistency in systems where different processors attempt to read and write from and to the same memory locations.
Scheduling The main goal is to assign processes to processors so that communication time and overhead are minimized.
Synchronization It is necessary to maintain the correct execution order by imposing the satisfaction of data dependencies.

Chapter 3

Vectorization and Fortran

3.1 Introduction

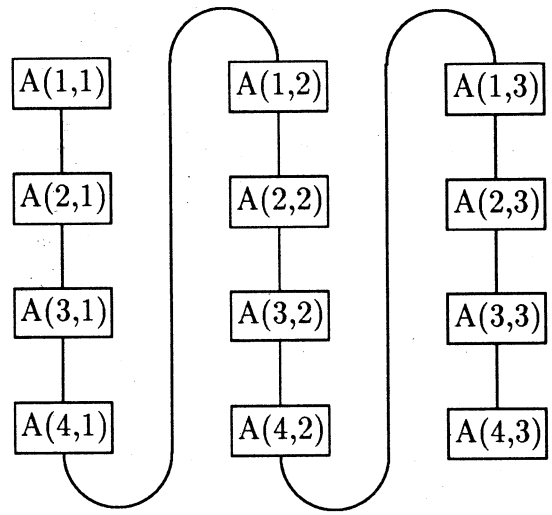
FORTRAN, created in the late 1950s first, is one of the most widely used programming languages for solving problems in science and engineering. A new standard has recently been finalized and the version of FORTRAN 90 has many new features. We will use FORTRAN as a model language in describing the details of vectorization.

3.2 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
```

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

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

then we have stride 4.

Exercise: What is the stride if we reference diagonal-wise:

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

3.3 Fortran 90

A new Fortran standard has been adopted 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 describe some new features, which are important for vector and parallel computers.

3.3.1 Vectors and Matrices

One of the most important new concepts in Fortran 90 is that arrays (vectors and matrices) are 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 a(1:m,1:n), b(1:m,1:n), c(1:m,1:n)
```

Arithmetic operations for matrices can be written

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

Matrix multiplication is **element-wise.**

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

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

gives as result a matrix `b`, the elements of which are

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

3.3.2 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. This can be used to write

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

which in Fortran 77 was written

```
do i=1,10
   x((i-1)*2+1)=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.**

There are variants:

```
i:j
```

which means that stride 1 is assumed. If the first variable is omitted:

```
:j
```

then the lower limit in the declaration of the array is assumed. The variant

```
:
```

means that both the lower and upper limit 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 the first of the following references

```
x(1:50,1:10)
x(:,25:30:2)
```

is a reference to the upper left submatrix of dimensions 50×10, and the second 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 appropriate dimensions. 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, and in understanding how array sections are executed using vector instructions.

Rule: In the assignment

```
array section=expression
```

the whole expression in the right hand side is computed before the assignment takes place (imagine that the operations are performed in vector registers).

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 equivalent 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).

3.3.3 Vector Mask Operations

The construct

```
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.

3.4 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.

3.4.1 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.

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

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

[±invariant expression *] CII [±invariant expression]

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

In the example

```
real w(100), x(100,50), y(50,1500,2), z(1000)
do i=1,n
   j=3*i+3-n
   k=1*j-5
   x(i,n)=y(m,5*j,1)+z(k-4)/w(i)
enddo
```

all the references to arrays are vector references:

array	s length	stride	
w	$w(1)$	n	1
x	$x(1,n)$	n	1
y	$y(m,30-5*n,l)$	n	750
z	$z((6-n)*l-9)$	n	$3*l$

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

3.4.2 Indirect Addressing

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

```
do i=1,64
   a(ia(i))=b(ib(i))+c(ic(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

```
a(ia)=b(ib)+c(ic)
```

Loading data this way (`b` and `c`) is called **gather**, and storing (`a`) is called **scatter**, cf. Section 2.2.4. 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.

3.4.3 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
   sca=a(i)*b(i)+sqrt(x(i)**2+y(i)**2)
   r(i)=sca*y(i)
   z(i)=(d(i)+e(i))/sca
enddo
```

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

```
V1=a(1:n)*b(1:n)+sqrt(x(1:n)**2+y(1:n)**2)
r(1:n)=V1*y(1:n)
z(1:n)=(d(1:n)+e(1:n))/V1
sca=[last element of V1]
```

3.4.4 Recursion

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

```
s=t+u
x=s*y
```

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
   a(i)=c(i)+b(ib(i))
   b(i)=x(i)*y(i)
   d(i)=e(i)/a(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. It is possible that for some previous `i`, `b(ib(i))` has been modified (e.g., if `ib(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 the programmer should give the compiler directives to vectorize (and he/she becomes responsible for errors, not the compiler).

Exercises:

1. Can the code

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

   be vectorized?

2. Let the vector of indices be `ia=(1,2,1)`. Can the code

```
do i=1,3
   a(ia(i))=a(ia(i))+b(i)
enddo
```

be vectorized?

3. Consider the matrix equation

$$BX = D,$$

where B is bidiagonal and all the matrices are $n \times n$ (for simplicity). This can be regarded as a sequence of bidiagonal systems

$$Bx_j = d_j, \quad j = 1, 2, \ldots, n,$$

where x_j and d_j are the column vectors X and D, respectively. The matrix equation can be solved by changing the algorithm in the text above so that each `x(i)` is replaced by `x(i,j)` and the corresponding for `d`, and by adding a `j` loop outside the `i` loop. Do this.

Then rewrite the code by exchanging the order of the loops. Does it vectorize now? If so, why?

3.4.5 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

```
sum=0.0
do i=1,n
   sum=sum+x(i)
enddo
```

or matrix multiplication

```
do j=1,n
   do i=1,n
      a(i,j)=0.0
      do k=1,n
         a(i,j)=a(i,j)+b(i,k)*c(k,j)
      enddo
   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
```

Exercise: Why are two vadd operations performed in the loop?

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.

Note, however, that if the scalar variable is used in more than one statement of the loop, then this can inhibit the use of such special instructions.

```
sum=0.0
do i=1,n
   sum=sum+x(i)*y(i)
   z(i)=sum*t(i)
enddo
```

Here the components of the vector z are functions of the partial sums, and therefore all these must be made available.

3.4.6 Rounding Errors

The result of a computation executed by vector operations may not be exactly the same as when the computation is performed in scalar mode, since the individual arithmetic operations may be performed in different order.

Assume that we have a computer with 15 decimal digits of accuracy in floating point representation. The code

```
a(2:10000)=1.0E-16
a(1)=1.0
sum=0.0
do i=1,10000
   sum=sum+a(i)
enddo
```

will give the result `sum=1`, if executed in scalar mode. If performed using vector instructions (using e.g. the code from the beginning of the preceding section) it is likely that the result is different. By summing the elements in this order, enough small elements will be added so that some partial sums are greater than 10^{-15}, and thus will give a noticeable contribution when added to 1.

3.4.7 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.)

Chapter 4

Numerical Algorithms on Vector Computers

4.1 Matrix Multiplications

4.1.1 Matrix–vector Product

To illustrate how linear algebra computations should be organized to execute efficiently on vector computers, we use the example of matrix–vector product $y = Ax$, where A has dimension $m \times n$. The components of y are given by

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

The algorithm

```
{* ij variant *}
do i=1,m
   y(i)=0
   do j=1,n
      y(i)=y(i)+a(i,j)*x(j)
   enddo
enddo
```

performs this computation. Symbolically we can write:

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

This figure should be interpreted as follows: each element of the right hand side vector is equal to the inner product of x and one row of the matrix A.

This algorithm has a couple of disadvantages:

1. The elements of A are referenced row-wise, which means that there is a risk of memory bank conflicts.

2. Even though inner products can be vectorized (using special instructions and hardware), they are usually slower than "real" vector operations.

We can exchange the order of the loops by writing the matrix as a collection of column vectors

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

we can write the multiplication in the form

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

This is done by the code

```
{* ji-variant *}
do i=1,m
   y(i)=0
enddo

do j=1,n
```

```
   do i=1,m
      y(i)=y(i)+a(i,j)*x(j)
   enddo
enddo
```

The inner loop is a **Saxpy** operation: **a vector plus a scalar times a vector** (S denotes single precision). The code can be written using vector assignments:

```
{* ji-variant *}
y(1:m)=0

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

Symbolically:

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

Here we have vector operations, which can be chained and executed very efficiently.

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 than the standard word length, the conversion must take place, and this is usually done by storing y in the primary memory. Since we are only interested in the final value of y, there are $n-1$ unnecessary `vstore` operations.

Instead we would prefer to accumulate y in a vector register. This variant is sometimes called **Gaxpy** (Generalized Saxpy). (In the code below we assume that a scalar can be stored in the multiplication unit and used in a vector operation.)

```
{* Gaxpy-variant *}
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)
```

Exercises

1. Which of the above variants is more efficient when $m << n$?

2. How should the matrix–vector multiplication be organized if m is much larger than the length of the vector registers?

Compilers will usually recognize when a chaining operation can be done. Unfortunately, compilers do not always recognize when intermediate results could stay in vector registers nor when the overlapping of loads and arithmetic operation can be done. To simulate Gaxpy, **loop unrolling** can be done. For simplicity, assume that n is a multiple of 4. The ji-variant code can then be written

```
{* ji-variant, 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 is only necessary to store it in primary memory when j changes.

4.1.2 Matrix Multiplication

Consider the problem of computing the matrix product

$$A = BC,$$

where, for simplicity, we assume that all matrices are square with order n.

A common matrix multiplication algorithm can be written as

```
{* Sdot variant *}
do i=1,n
   do j=1,n
      a(i,j)=0
      do k=1,n
         a(i,j)=a(i,j)+b(i,k)*c(k,j)
      enddo
   enddo
enddo
```

In the same manner as we did with matrix–vector multiplication, we can now change the order of the loops in $3! = 6$ different ways. The above variant is based on scalar products (Sdot).

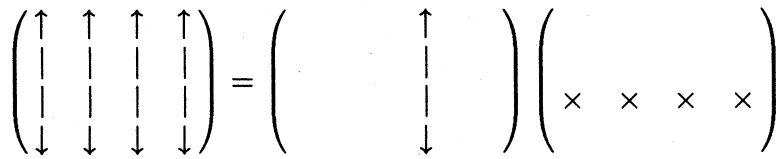
Disregarding the zero initialization of A, we can write the generic matrix multiplication algorithm

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

If we restrict ourselves to column oriented variants without scalar products, two variants are left (we denote them by the order of the indices): `kji` and `jki` (note that by making `i` the last index, we have column oriented vector operations in the innermost loop).

We can illustrate the computation of the two innermost loops symbolically:

`kji :`

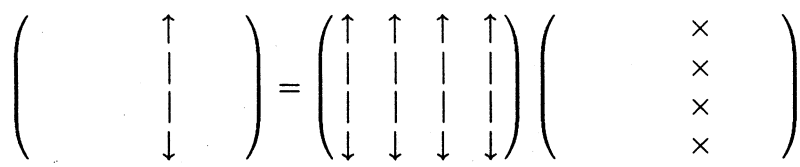


The figure should be interpreted: "each column of A is updated by adding a multiple of a column from B".

We see that the operation is Saxpy, and that each column of A must be fetched from primary memory, updated and then stored back.

The two innermost loops of the `jki` variant can be illustrated:

`jki :`



This should be interpreted: "a column from A is computed by summing multiples of the columns of B".

This variant can be implemented using Saxpy operations, but we see that since the computation of each column of A is finished before it is stored back, we have a Gaxpy oriented algorithm. Furthermore, since the columns of B need only be loaded from primary memory, and not stored back, we have only half as much memory traffic as in the `kji` variant.

The memory traffic in the three variants described above, Sdot (`ijk`), `kji` and `jki`, is summarized in Table 4.1.2 (only the highest order term is given).

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

Table 4.1: Memory traffic in matrix multiplication.

4.2 BLAS: Basic Linear Algebra Subprograms

BLAS consists of a number of subroutines for basic linear algebra computations. The first level of BLAS were developed for LINPACK, which is a library of subroutines for the solution of linear systems of equations.

One of the main reason for developing 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 all different computers without changes.

There are BLAS routines in single precision (with prefix S, e.g. Saxpy), double precision (Daxpy), complex single precision (Caxpy), and complex double precision (Zaxpy).

Level 1 BLAS

The first level of BLAS routines are vector–scalar operations and vector–vector operations (e.g. Saxpy and Sscal; the latter scales a vector by a scalar).

As an example we take Snrm2, whose declaration begins

```
real function snrm2(n,sx,incx)
```

where `n` is the number of elements of the vector `sx`, the norm of which is to be computed, and `incx` is the increment (stride) in the vector. 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 *}
```

We list the level 1 BLAS in the following where x and y are vectors:

```
call _COPY(n,x,incx,y,incy)        overwrite y with x
     S,D,C,Z

call _SWAP(n,x,incx,y,incy)        interchange contents of x and y
     S,D,C,Z

call _SCAL(n,a,x,incx)             x <- ax
     S,D,C,Z,   CS,ZD (complex vector, real scalar)
```

```
call _AXPY(n,a,x,incx,y,incy)      y <- ax + y
     S,D,C,Z

w = _DOT_(n,x,incx,y,incy)         <x, y>
    S,D,C,Z
    C(conjugate),U(unconjugate)

sw = _NRM2(n,x,incx)               2 norm of x
     S,D,SC,DZ

sw = _ASUM(n,x,incx)               1 norm of x
     S,D,SC,DZ

imax = _AMAX(n,x,incx)             index of the component of maximum
       IS,ID,IC,IZ                 absolute value

call _ROTG(a,b,c,s)                returns c and s such that
     S,D                           [c s;-s c]*[a;b] = [r;0]

call _ROT(n,s,incx,y,incy,c,s)     [x y] <- [x y]*[c -s;s c]
     S,D,CS,ZD
```

Level 2 BLAS

When computers with vector instructions became available, it was soon evident that Level 1 BLAS did not utilize the hardware efficiently (cf. the discussion of Saxpy and Gaxpy above). This lead to the definition of a set of routines on a higher level, and thus the Level 2 BLAS routines perform matrix–vector operations.

Some examples of Level 2 BLAS operations are

- matrix–vector multiplication, $y = \beta y + \alpha Ax$, 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). The discussion above about Saxpy and Gaxpy operations shows that much can be gained by implementing matrix–vector multiplication in assembler language.

- outer product, $A = A + \alpha x y^T$, where x and y are column vectors. The subroutine is called Sger ("r" denotes "rank 1 update").

- solution of a system $x = T^{-1}x$, where the matrix T is triangular. The subroutine is called Strsv ("tr" for triangular and "sv" for solve).

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 matrix–matrix operations. Typical examples are

- Matrix product $A = \beta A + \alpha BC$. The subroutine is called Sgemm ("ge" for general matrix, "mm" for matrix multiplication).

- Solution of triangular systems with several right hand sides $B = T^{-1}B$. The subroutine is Strsm ("sm" for the solution of system with multiple right hand sides).

BLAS and memory references

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.

Rule: For each memory reference, perform as many floating point operations as possible.

We will now consider the different levels of BLAS from the point of this rule. 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 load 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 multiplication, Sgemm, loads three matrices and stores one, altogether $4n^2$ memory references. Here we have $2n^3$ flops.

We summarize in Table 4.2, where we also give Megaflop rates on three high performance computers. The data are taken from [6].

BLAS level	routine	ref.	flops	flops/ref	Cray-2 peak 488	IBM 3090VF peak 108	Alliant FX/8 peak 88
1	Saxpy	$3n$	$2n$	2/3	121	26	14
2	Sgemv	n^2	$2n^2$	2	350	60	26
3	Sgemm	$4n^2$	$2n^3$	$n/2$	437	80	43

Table 4.2: Memory traffic and floating point operations. Megaflop rates for three computers. Unfortunately the source did not show the size of the problems.

It is very simple 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, the solution of triangular systems with multiple right hand sides are independent and can be distributed to different processors.

4.3 Linear Systems of Equations

4.3.1 Gaussian Elimination and LU Decomposition

Suppose we would like to solve a linear system of equations

$$Ax = b,$$

where the matrix A has order n, and is assumed to be non-singular (see [9]).

For simplicity we will not consider pivoting in the algorithm for Gaussian elimination. After $k-1$ steps in the algorithm we have reduced the matrix to

the form

$$\begin{pmatrix} a_{11} & a_{12} & & & \cdots & & a_{1n} & b_1 \\ & a_{22} & & & \cdots & & a_{2n} & b_2 \\ & & \ddots & & & & \vdots & \vdots \\ & & & a_{kk} & a_{k,k+1} & \cdots & a_{kn} & b_k \\ & & & \vdots & \vdots & & \vdots & \vdots \\ & & & a_{ik} & a_{i,k+1} & \cdots & a_{in} & b_i \\ & & & \vdots & \vdots & & \vdots & \vdots \\ & & & a_{nk} & a_{n,k+1} & \cdots & a_{nn} & b_n \end{pmatrix}.$$

In the next step of the elimination the elements below the main diagonal in column k will be annihilated. The result is

$$\begin{pmatrix} a_{11} & a_{12} & & & \cdots & & a_{1n} & b_1 \\ & a_{22} & & & \cdots & & a_{2n} & b_{2n} \\ & & \ddots & & & & \vdots & \vdots \\ & & & a_{kk} & a_{k,k+1} & \cdots & a_{kn} & b_k \\ & & & \vdots & \vdots & & \vdots & \vdots \\ & & & 0 & a'_{i,k+1} & \cdots & a'_{in} & b'_i \\ & & & \vdots & \vdots & & \vdots & \vdots \\ & & & 0 & a'_{n,k+1} & \cdots & a'_{nn} & b'_n \end{pmatrix},$$

where the transformed elements are given by

$$\begin{aligned} a'_{ij} &= a_{ij} - a_{ik}a_{kj}/a_{kk}, \qquad j = k+1,\ldots,n, \quad i = k+1,\ldots,n, \\ b'_i &= b_i - a_{ik}b_k/a_{kk}, \qquad i = k+1,\ldots,n, \end{aligned}$$

Thus Gaussian elimination can be performed by the following program (for simplicity we omit the transformations of the right hand side vector b):

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

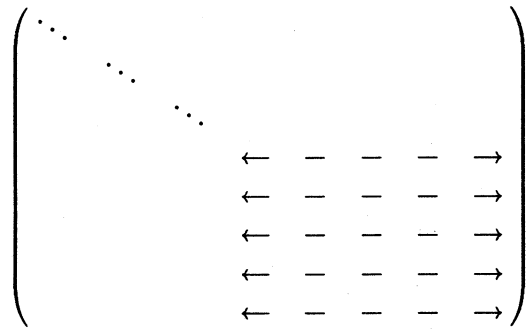
As in the matrix multiplication, we can change the order of the loop and describe Gaussian elimination by the following generic algorithm:

```
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 the different alternatives depends on the architecture.

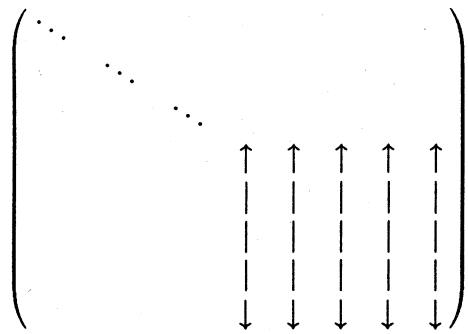
We discuss three variants. First consider the `kij` variant (in the literature it is often called the text book variant). Here the matrix is referenced as follows in the two innermost loops:

`kij` :



As in the case of matrix multiplication, the `kji` variant is a Saxpy oriented algorithm:

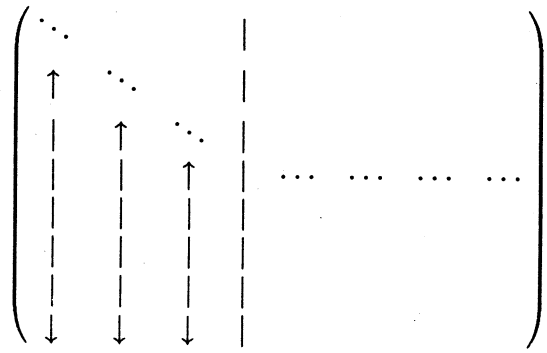
`kji` :



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

The Gaxpy variant, alternatively the `jki` 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 zeroed, 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**.

In the same way as matrix multiplication, the most efficient variant on vector computers, e.g., Cray Y-MP, should be the one with as few memory references as possible (all have the same number of flops). In Table 4.3.1 we summarize the number of memory references for the three variants considered.

kij	kji	jki
$2n^3/3$	$2n^3/3$	$n^3/3$

Table 4.3: Memory references in three variants of Gaussian elimination.

4.3.2 Block Algorithms for LU Decomposition

In Section 4.2 we indicated that one can maximize the number of flops per memory reference by organizing the computations as block algorithms. For Gaussian elimination (LU decomposition) this can be done in the following way.

Consider the matrix A and partition it in blocks. For simplicity, we consider the case of two blocks.

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

Assume that A_{11} is square and non-singular. A block LU decomposition can be computed from 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

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

from which we have

1. $L_{11}U_{11} = A_{11}$. We can compute L_{11} and U_{11} by the usual Gaussian elimination algorithm applied to A_{11}

2. $L_{11}U_{12} = A_{12}$. Given L_{11} from step 1 we can compute U_{12} by solving a number of lower triangular systems.

3. $L_{21}U_{11} = A_{21}$. Given U_{11} from step 1 we can compute L_{21} by solving a number of upper triangular systems.

4. $S_{22} = A_{22} - L_{21}U_{12}$. The matrix S_{22} can be stored in the same place as A_{22}. This step can be organized as updating the $(2,2)$ block: $A_{22} := A_{22} - L_{21}U_{12}$.

Obviously we can now continue the procedure with the block A_{22}, and we can formulate an algorithm for **block LU decomposition**. For simplicity, assume that the matrix dimension n satisfies $n = t * nb$, for some integers t and nb.

```
* Right-looking block LU decomposition

* Pivoting is omitted

do i=1,t
```

```
s=(i-1)*nb+1                          *Start position of block to decompose
e=i*nb                                *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
```

The memory references are illustrated in Figure 4.1.

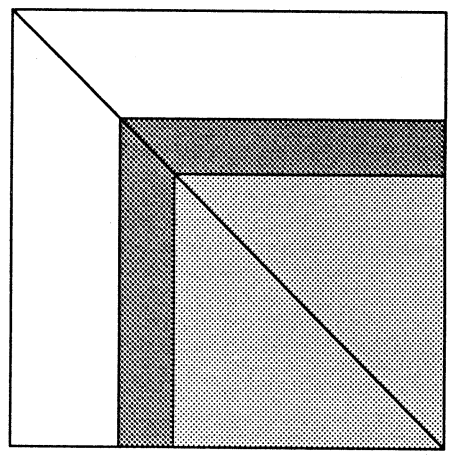


Figure 4.1: Right-looking block LU decomposition. The darker shade indicates the elements that are computed in the present block transformation and the lighter shade indicates elements that are updated.

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

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & U_{33} \end{pmatrix}, \qquad (4.1)$$

and assume that we have performed the first block step and U_{11}, L_{11}, L_{21} and L_{31} are known, 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 (4.1) we get

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

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

$$U_{12} := L_{11}^{-1}A_{12}.$$

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 code, and compute $L_{32} := A_{32}U_{22}^{-1}$.

The algorithm is

```
* Left-looking block LU decomposition.  Pivoting is omitted *

do i=1,t

     s=(i-1)*nb+1              * Start position of block to decompose
     e=i*nb                    * 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,s:e} U_{1:s-1,s:e}   * SGEMM
     L_{s:n,s:e} U_{s:e,s:e} = A_{s:n,s:e}                         * BLAS-2

enddo
```

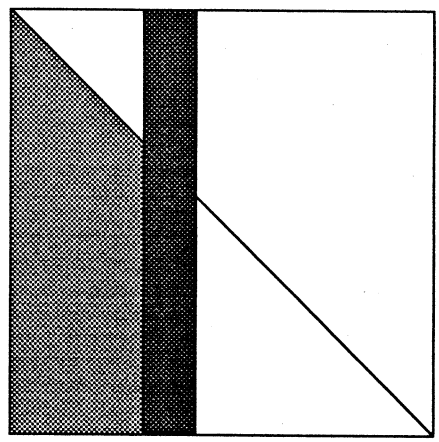


Figure 4.2: Left-looking block LU factorization. Lighter shade indicates elements that are used and darker shade indicates elements computed in the present stage.

The memory reference pattern is illustrated in Figure 4.2.

The performance of two variants of block LU decomposition is illustrated in Table 4.3.2 (data from [2]).

Note that the main part of the work in block LU decomposition is done in BLAS-3 routines, depending on block size. The variants differ in how much of the work is done by which subroutines. A subroutine may execute 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.

4.3.3 LAPACK

LAPACK [1] 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 (linear systems of equations and least squares) and the EISPACK library (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

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

same time they perform well on most computers, in particular if the BLAS routines have been optimized for each computer. On supercomputers 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. Therefore, the LAPACK routines parallelize automatically.

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.

LAPACK was developed by an international group of researchers and it is available at no charge by electronic mail through *netlib*, at one of the e-mail addresses

```
netlib@ornl.gov
netlib@research.att.com
```

To obtain an index of the library send mail to one of the above addresses with the message

```
send index from lapack.
```

Chapter 5

Numerical Algorithms on Distributed-memory Computers

5.1 Load-Balancing for Matrix Computations

Load balancing is important in achieving high efficiency on parallel computers. The initial mapping of the data should be carefully designed by analyzing the load balancing of a given algorithm before execution, and taking corrective measures. A more difficult case arises when the behavior of the algorithm is not predictable before execution because of the dynamic nature of the algorithm. For example, in the block Jacobi method for the eigenvalue decomposition, it is often the case that some blocks have reached numerical convergence while others are still going through iterations. After a number of iterations, the balance in work-load can become very poor and only dynamic load balancing will remedy the situation. Another example where more extreme case can be observed is the finite element codes that rely on dynamic grid refinement.

5.1.1 Mapping Matrices to Processors

Before a parallel processor can start a computation, the matrix has to be partitioned so that parts of it can be assigned to different processors. The way the data is distributed among the processors has enormous impact on the performance of a parallel system and it is important to analyze for the most

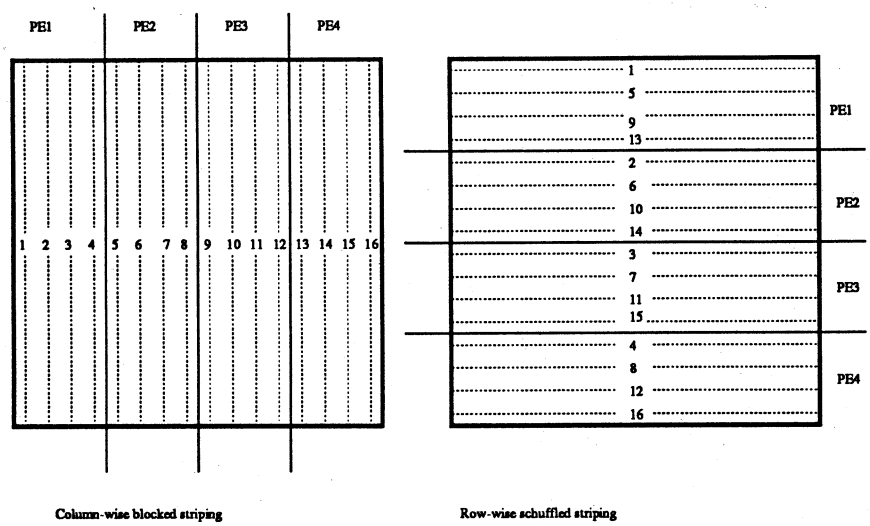


Figure 5.1: Striped mapping

efficient data mapping for a particular algorithm.

Striped Mapping

In a striped partitioning, a matrix is divided into groups of rows or columns and each group is assigned to a processor. The partitioning is **blocked** if each processor has contiguous rows or columns. If rows or columns are distributed among the processors so that processor i contains columns $i, i+p, i+2p \cdots$, then it is called **cyclic** mapping (alternatively wrap-around, interleaved, shuffled, etc. We have chosen to use the terminology of High Performance Fortran (HPF, [12])). The matrix can also be partitioned using a method which is a combination of these two mappings.

The left mapping in Figure 5.1 is achieved in HPF by the following compiler directive.

```
!HPF$ Distribute a(*,block)
```

The second dimension of `a` (i.e., the rows) is divided up into blocks, which are distributed to the 4 processors. The `*` indicates that the matrix will not be distributed along its first dimension (columns).

The cyclic mapping on the right in Figure 5.1 is specified in the following directive

11 12 21 22	13 14 23 24	15 16 25 26	17 18 27 28
31 32 41 42	33 34 43 44	35 36 45 46	37 38 47 48
51 52 61 62	53 54 63 64	55 56 65 66	57 58 67 68
71 72 81 82	73 74 83 84	75 76 85 86	77 78 87 88

Figure 5.2: Blocked checkerboard mapping. Each square represents a processor.

```
!HPF$ Distribute a(cyclic,*)
```

The rows are distributed cyclically over the 4 processors.

Checkerboard Mapping

In Checkerboard mapping, the matrix is partitioned into square or rectangular submatrices and distributed among processors, see Figure 5.2. As in striping, checkerboard mapping can be blocked or cyclic (shuffled). This mapping fits naturally onto a two dimensional array of processors. The finest grain in this mapping is to assign one matrix element to one processor, which means that it can potentially exploit more concurrency than striping.

This distribution is specified as follows

```
!HPF$ Distribute a(block,block)
```

5.2 Message-Passing Systems

Message passing parallel computer systems are usually of MIMD type. Communication of data is done by sending messages from one processor to another (or to several), using the interconnection network. A message is sent or received by making a call to a special communication routine, `send` or `receive`.

5.2.1 Matrix Multiplications

We describe briefly parallel versions, in a message passing context, of the SAXPY and SDOT (see the preceding chapter) variants of matrix-vector multiplication.

First, we consider the matrix-vector multiplication $y = Ax$ where A is $m \times n$. We can write

$$y = \sum_{i=1}^{n} a_i x_i,$$

where a_i denotes the i'th column of A. If, for simplicity, we assume that the number of processors p is the same as n and that a_i, and i'th component of x, x_i, are assigned to processor i, then all the products $x_i a_i$ can be computed in parallel, cf. Figure 5.3.

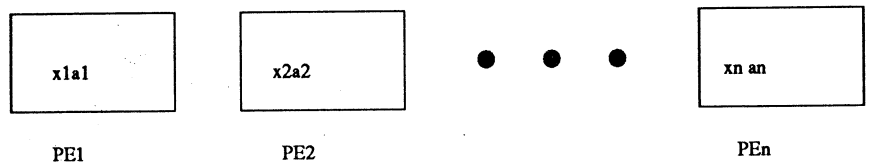


Figure 5.3: Parallel matrix-vector multiplication algorithm by linear combination

To compute y we need to add up all the vectors and this is done by a fan-in algorithm, which requires transfer of data. Synchronization is needed to ensure that the necessary multiplications are complete before addition begins and this can be performed 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.

Assume that $p = n = 2^d$, and that the processors are numbered $1, 2, \ldots, p$. Each processor is further assumed to have a local variable, `myid`, which is its id 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(vector,proc-number)` and `receive(vector)`, where the latter means that a message from any other

processor is received. The whole computation is done if each processor executes the following code.

```
%
% Parallel Matrix-vector multiplication.
%
% Each processor has local variables ai (vector) and xi (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
%
y=xi*ai(1:m)
if (myid/2)*2+1 = myid then
   send(y,myid+1)              % Even processors will continue
else
   do k=1,d                    % Fan-in algorithm
      twok=2**k
      if (myid/twok)*twok = myid then
         receive(y1)
         y=y+y1
         twok1=2**(k+1)
         if (myid/twok1)*twok1 not= myid then
            send(y,myid+twok)
         endif
      endif
   enddo
endif
```

Exercise: Which processor holds the final result in the above algorithm?

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

The SDOT (inner product) variant of matrix vector multiplication is de-

rived from the expression

$$y = Ax = \begin{pmatrix} \overline{a}_1^T \\ \overline{a}_2^T \\ \vdots \\ \overline{a}_m^T \end{pmatrix} x = \begin{pmatrix} \overline{a}_1^T x \\ \overline{a}_2^T x \\ \vdots \\ \overline{a}_m^T x \end{pmatrix},$$

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

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.

For matrix-matrix multiplications, other than the algorithms that can be obtained easily by generalizing the matrix-vector product algorithms, there are also algorithms based on the checkerboard partitioning. Assume that we want to compute $C = AB$ and the matrices are partitioned commensurately as

$$C = AB = \begin{pmatrix} A_{11} & \cdots & A_{1r} \\ \cdot & \cdots & \cdot \\ \cdot & \cdots & \cdot \\ \cdot & \cdots & \cdot \\ A_{s1} & \cdots & A_{sr} \end{pmatrix} \begin{pmatrix} B_{11} & \cdots & B_{1t} \\ \cdot & \cdots & \cdot \\ \cdot & \cdots & \cdot \\ \cdot & \cdots & \cdot \\ B_{r1} & \cdots & B_{rt} \end{pmatrix} = \Sigma A_{ik} B_{kj}.$$

This leads to a variety of algorithms. For example, if $p = s * t$, the number of blocks in C, then each block can be computed in parallel after proper distribution of matrix elements. If $s = t = 1$, then it gives block inner product algorithm and when $r = 1$, we have block outer product algorithm.

5.2.2 Gaussian Elimination

To further illustrate the programming of a distributed memory parallel computer with message-passing, we consider Gaussian elimination for solving a linear system of equations

$$Ax = b,$$

where A is an $n \times n$ matrix, and n is large. We will study the implementation detail of a specific algorithm in this section. A variety of parallel algorithms

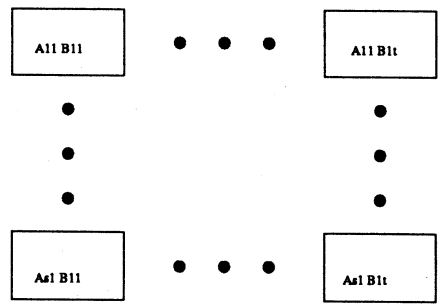


Figure 5.4: Parallel block multiplication

for Gaussian elimination can be developed in a way that is analogous to those methods for matrix multiplication we discussed in the previous section.

The code for solving the system can be written (for simplicity we do not do pivoting)

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

We rewrite the code as

```
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+1:n,k)*a(k,j)
   enddo
enddo
```

and, we rewrite it again to obtain

```
do k=1,n-1
   cdiv(a(k+1:n,k),a(k,k))
   do j=k+1,n
      saxpy(a(k+1:n,j),a(k+1:n,k),a(k,j))
   enddo
enddo
```

where `cdiv` and `saxpy` are procedures defined by

```
cdiv(x,s)
   x=x/s

saxpy(y,x,s)
   y=y-s*x ,
```

`x` and `y` are vectors of the same length, and `s` is a scalar.

Now, we implement this algorithm on a ring of p processors. Since we will

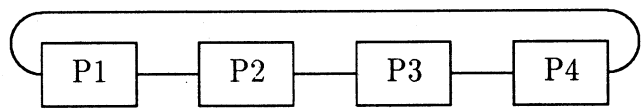


Figure 5.5: Ring of processors, $p = 4$.

only consider neighbor-to-neighbor communication, we can restrict ourselves to the following communication primitives:

```
send(east,x)
```

which means that the vector `x` is sent to the neighbor to the east of the present processor. Note that the processor to the east of P4 is P1.

```
receive(west,x)
```

means that `x` is received from the neighbor in the west. The processors are numbered from 1 to p, and each processor is assumed to have stored its number in the variable `myno`.

Since the algorithm is column based, we will consider assignments of columns to the different processors. First assume that the matrix A is divided into blocks of equal size,

$$A = (A_1 \quad A_2 \quad \cdots \quad A_p) .$$

Then, since we are transforming A to upper triangular form, we see that with this assignment processor 1 becomes idle after the initial n/p steps, then after n/p steps additional steps processor 2 becomes idle, etc. Obviously, this assignment will not give a good load balancing. Instead, we distribute the columns to the processors in an cyclic way, as illustrated in Figure 5.6.

The elimination is performed if each processor executes the following program.

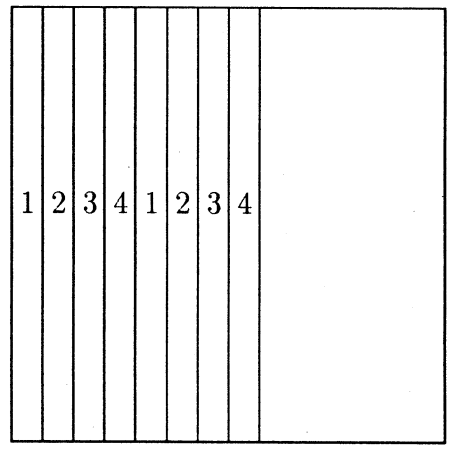


Figure 5.6: Cyclic assignment of columns to processors, $p = 4$.

```
% Dimensions: n=r*p.
% The columns of the local matrix C(1:r) are columns
%    A(:,myno), A(:,p+myno), A(:,2p+myno),...
%

j=0
do k=1,n
   if mod(k,p)=myno then      % Remainder when k is divided by p
                              % This processor holds the
                              % pivot column
      j=j+1
      cdiv(C(k+1:n,j),C(k,j))
      send(east,C(k+1:n,j),p)
      piv_col(k+1:n)=C(k+1:n,j)
   else
      receive(west,piv_col(k+1:n),counter)
      counter=counter-1
      if counter>1 then
         send(east,piv_col(k+1:n),counter)    % Immediately
                                              % send it further
      endif
   endif
```

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

We use the variable `counter` to keep count of how many processors 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 processor 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.

Partial pivoting is needed in the Gaussian elimination algorithm for numerical stability. When A is stored by column cyclic mapping, as above, then the search for the pivot element takes place in one processor. In the meantime all other processors are idle. Information about the pivoting can be sent to all processors along with the pivot column, and then the interchange of rows can be done in parallel within all the processors, before the SAXPY operations are performed.

5.2.3 Solution of Triangular Systems

After the Gaussian elimination, we need to solve triangular systems. We will consider only upper triangular systems $Ux = c$ since the lower triangular case is similar and also we can update the right hand side vector simultaneously as the triangularization process and there is no need to solve the lower triangular system. We assume that U is stored by column wrapped storage, then the following column sweep algorithm can be implemented in a straightforward way.

```
do j=n,1,-1
   x(j) = c(j)/U(j,j)
   do i=1, j-1
      c(i) = c(i) - x(j) * U(i,j)
   enddo
enddo
```

Assume that P_n, the processor holding the last column of U also holds c. Then P_n computes x_n, updates c, and sends the new c to P_{n-1}. Then P_{n-1}

computes x_{n-1}, updates c and sent to P_{n-2}. Thus, only one processor is doing computation at any given time.

The inner product algorithm is

```
do i = n,1,-1
   do j=i+1,n
     c(i) = c(i) - U(i,j)*x(j)
   enddo
   x(i) = c(i)/U(i,i)
enddo
```

This algorithm uses the inner product of ith row excluding the main diagonal of U with $x(i+1:n)$ in the computation of x_i. We assume that these components have been computed so that x_j is in P_j and P_j can compute $\Sigma u_{ij}x_j$ for all the x_j that it holds. After these partial inner products are computed in parallel, fan-in can be used to sum them. For large n and i, there is almost perfect parallelism in computing partial inner products although fan-in is less satisfactory.

```
% parallel inner product algorithm*

do i=n,1,-1
   all processors compute their portion of i'th inner product
   fan-in partial inner products to P(i)
   P(i) computes x(i)
enddo
```

Exercise: Design a parallel algorithm for the Gaussian elimination and triangular system solving as shown in this and previous sections assuming that the matrix is distributed by rows over the processors. Discuss the advantages and disadvantages over the column oriented method.

5.3 Data-Parallel Computations

In data-parallel algorithms the basic idea is that the processors of the parallel computer are assigned to the elements in a matrix in such a way that each element has its own processor, where it is stored in the local memory, and where the computations take place. Thus, the matrix is the basic structure, and the computer is organized to match this structure. We assume that the computer has communication links that make up a two-dimensional array, and that the matrix elements in A are stored as in Figure 5.7.

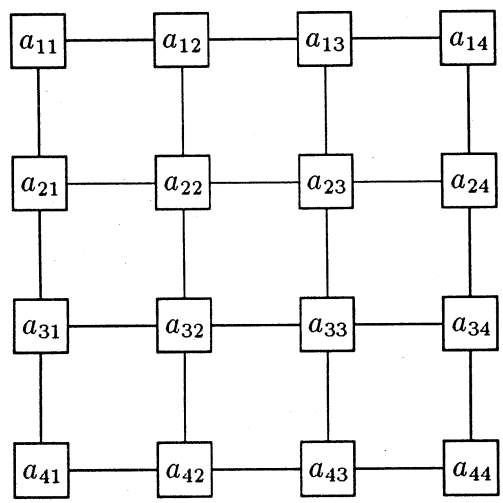


Figure 5.7: Array of processors and matrix elements.

5.3.1 Distributed-Shared Memory

In writing a data-parallel program, we are not explicitly concerned with the communication aspects of the computation. Thus we can write

```
A(1:10,1:10)=A(1:10,1:10)+B(21:30,21:30)
```

Since `A(i,j)` is stored in the same processor as `B(i,j)`, it is necessary to move the array section from `B` to the processors that hold `A(1:10,1:10)`, perform the addition, and then store results in the local memories. This is hidden to the programmer and takes place automatically (the necessary communication code

is generated by the compiler). Thus, processor (i,j) executes the following code.

```
% Processor (i,j)
% Local variable a corresponds to A(i,j)
load B(i+20,j+20) --> b_reg      % Request to the communication
                                 % system
a=a+b_reg
```

In the shared memory context the load instruction fetches a word from main memory to a register. Here, load is a request to the communication system to locate a variable in the local memory of another processor, and then fetch the value to the processor (i,j).

From the programmers point of view there is no difference between this aspect of the data-parallel program and the same program executed on a shared memory computer, since there is no need to deal with the communication explicitly. Therefore, in connection with data-parallel programming, we can refer to this memory organization as **distributed-shared**.

5.3.2 Gaussian Elimination

To develop a data-parallel program for Gaussian elimination, we begin by writing the code in the following way:

```
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
```

Our aim is to reformulate this code in terms of data-parallel matrix assignments. The main part of the work in the algorithm is in the j,i-loop. We consider the communication needed for modifying the (i,j) element. The formula is

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

Thus, in order for the (i,j) processor to modify its element, it must have `a(i,k)` and `a(k,j)`. This communication is illustrated in Figure 5.8.

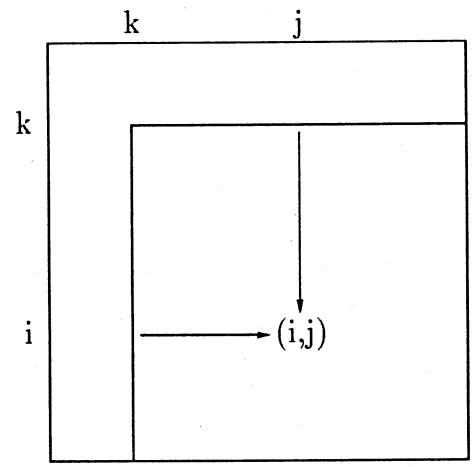


Figure 5.8: Communication for modifying `a(i,j)`.

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`. Thus

```
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 5.9.

The following code is Fortran 90 style Gaussian elimination.

```
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
```

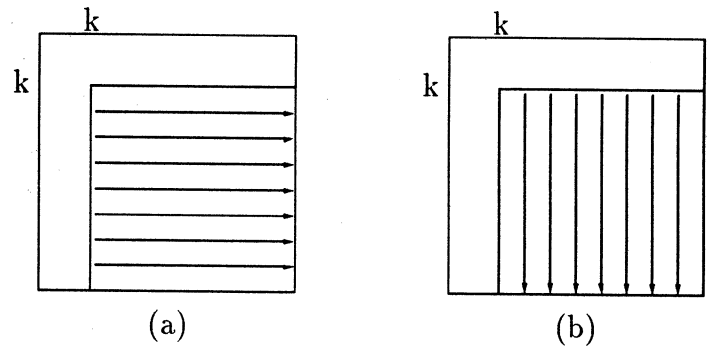


Figure 5.9: (a) `spread(a(k+1:n,k),2,n-k)` (b) `spread(a(k,k+1:n),1,n-k)`.

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 checkerboard fashion, see Figure 5.2. Then after a few steps in the algorithm, some processors will be idle (Figure 5.10). In order to achieve better load balancing we used a column

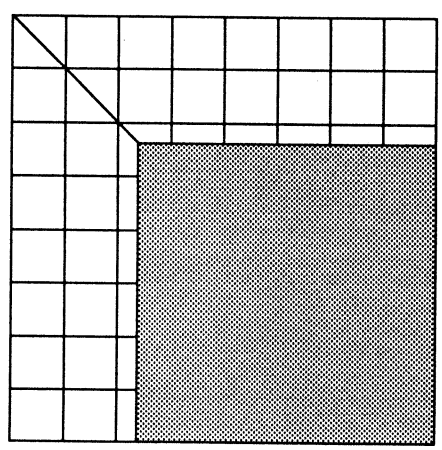


Figure 5.10: Bad load-balancing in the Gaussian elimination algorithm with blocked mapping. Each square represents a matrix block stored in one processor. The processors completely outside the shaded area are idle.

cyclic assignment in the message passing algorithm. Here we should use a block cyclic assignment as shown in Figure 5.11.

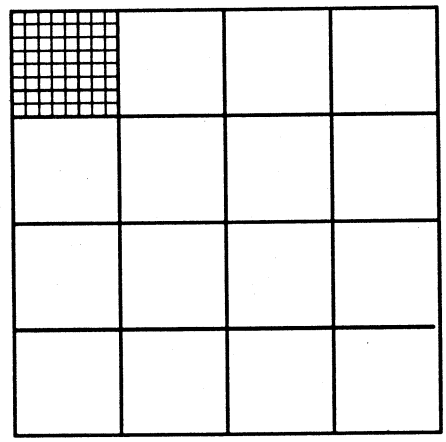


Figure 5.11: Block cyclic storage. The large squares represent matrix blocks. Each block is distributed over the whole array of processors.

This assignment can be made in HPF by the following compiler directive:

```
!HPF$ Distribute a(cyclic,cyclic)
```

and the data-parallel code combined with this directive will execute with good load-balancing.

5.3.3 Matrix Multiplication – Cannon's Algorithm

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

$$c_{11} = \sum_{k=1}^{4} a_{1k} b_{k1} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41}$$

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

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 5.12(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 5.12(b), and a second term in each sum can be computed. The algorithm proceeds as illustrated in Figure 5.12(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.

A11 B11	A12 B22	A13 B33	A14 B44
A22 B21	A23 B32	A24 B43	A21 B14
A33 B31	A34 B42	A31 B13	A32 B24
A44 B41	A41 B12	A42 B23	A43 B34

(a) Initial alignment of submatrices

A12 B21	A13 B32	A14 B43	A11 B14
A23 B31	A24 B42	A21 B13	A22 B24
A34 B41	A31 B12	A32 B23	A33 B34
A41 B11	A42 B22	A43 B33	A44 B44

(b) After the first shift

A13 B31	A14 B41	A11 B13	A12 B24
A24 B41	A21 B12	A22 B23	A23 B34
A31 B11	A32 B22	A33 B33	A34 B44
A42 B21	A43 B32	A44 B43	A41 B14

(c) After the second shift

A14 B41	A11 B12	A12 B23	A13 B34
A21 B11	A22 B22	A23 B33	A24 B44
A32 B21	A33 B32	A34 B43	A31 B14
A43 B31	A44 B42	A41 B13	A42 B24

(d) After the third shift

Figure 5.12: Four steps of Cannon's algorithm on 4×4 processors

```
subroutine cannon(A,B,C,N)
integer N
real, array(N,N)      ::A,B,C
% Perform the initial skewing of A and B
A=cshift(A,2,-(/0:N-1/))      % Horizontal
B=cshift(B,1,-(/0:N-1/))      % Vertical
C=0.0
do i=1,N
   C=C+A*B                    % Elementwise multiplication
   A=cshift(A,2,-1)           % Shift one step to the left
   B=cshift(B,1,-1)           % Shift one step up
enddo
return
```

The Fortran 90 intrinsic function `cshift`, which stands for *circular shift*, performs the communication necessary for this algorithm. In the code we have the statement `A=cshift(A,2,1)`, which specifies that A is to be shifted one step along the second dimension, i.e., rowwise, with wrap-around. In the statement `A=cshift(A,2,-(/0:N-1/))` each row of A is shifted by a different amount, given by the vector `/0:N-1/`. This means that the first row is shifted 0 positions, the second row 1 position, etc.

5.4 Jacobi Method for Eigendecomposition

A case study based on the paper, P.J. Eberlein and H. Park, Efficient implementation of Jacobi algorithms and Jacobi sets on distributed memory architectures, *Journal of Parallel and Distributed Computing*, special issue on *Algorithms for Hypercube Computers*, 8, pp. 358-366, 1990.

Bibliography

[1] E. Andersson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, LAPACK Users' Guide, SIAM, Philadelphia, 1992.

[2] E. Anderson and J. Dongarra, LAPACK Working Note 19, Evaluating Block Algorithms Variants In LAPACK, Computer Science Department, University of Tennessee, Knoxville, report CS-90-103, April, 1990.

[3] J. J. Dongarra, J. R. Bunch, C.B. Moler, and G.W. Stewart, LINPACK User's Guide, SIAM Publications, Philadelphia, 1978.

[4] K. Dackland, E. Elmroth, B. Kågström, C. Van Loan, Parallel Block Matrix Factorizations on the Shared Memory Multiprocessor IBM 3090 VF/600J, Int. J. Supercomputer Appl. 6(1992), 69-97.

[5] J. J. Dongarra, Performance of Various Computers Using Standard Linear Equations Software, Mathematical Sciences Section, Oak Ridge national Laboratory, Oak Rodge, TN 37831, Report CS-89-85, May 31,1993.

[6] J.J. Dongarra, J. Du Croz, I. Duff, S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs, Mathematics And Computer Science Division, Argonne National Lab., Preprint No. 2, August 1988.

[7] J.J. Dongarra, I.S. Duff, D.C. Sorensen, H.A. van der Vorst, Solving Linear System on Vector and Shared Memory Computers, SIAM, Philadelphia, 1991.

[8] L. Eldén and L. Wittmeyer-Koch, Numerical Analysis, An Introduction, Academic Press, San Diego, 1990.

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

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

[11] R.W. Hockney and C.R. Jesshope, Parallel Computers 2, Adam Hilger, Bristol, 1988.

[12] High Performance Fortran Language Specification, High Performance Fortran Forum, Rice University, Houston Texas, Version 1.0, May 1993.

[13] J. M. Levesque and J. W. Williamson, A Guidebook to Fortran on Supercomputers, Academic Press, San Diego, 1989.

[14] J.M. Ortega, Introduction to Parallel and Vector Solution of Linear Systems, Plenum Press, New York, 1988.

[15] Y. Saad, Unpublished manuscript on numerical parallel algorithms, 1992.