Analysis and design of parallel algorithms

ANALYSIS AND DESIGN OF PARALLEL ALGORITHMS

by

Richard Charles Dunbar, B.Sc.

A Doctoral Thesis

Submitted in partial fulfilment of the requirements

for the award of Doctor of Philosophy

of the Loughborough University of Technology

July, 1978.

Supervisor: Professor D.J. Evans, Ph.D.,D.Sc.

Department of Computer Studies

DECLARATION

I declare that the following thesis is a record of research work carried out by me, and that the thesis is of my own composition. I also certify that neither this thesis nor the original work contained therein has been submitted to this or any other institution for a degree.

R.C. DUNBAR.

ACKNOWLEDGEMENTS

I would first like to thank my supervisor Professor Evans without who's patience, guidance, encouragement, useful ideas and general shove in the right direction when needed, this thesis would never have been completed.

I am also indebted to Doctors Blakemore, Barlow and, in particular, Rick for their comments and assistance, and Miss Briers, who typed this thesis so professionally.

Finally, I wish to thank my family. My wife, who had to live with me and my parents who have always encouraged me. I dedicate this thesis to them and also my grandmother who saw it started and son who saw it finished. Thank you.

CONTENTS

Chapter 1

CURRENT PARALLEL COMPUTER ARCHITECTURES

1.1 INTRODUCTION

Since their introduction, the computation speed of electronic computers has been greatly increased mainly by the development of faster electronic components. The first computers used relatively slow components such as vacuum tubes and their central memories were magnetic drums. As electronic technology advanced, these components have been replaced by transistors and magnetic cores which in their turn have been replaced by integrated components.

The present state of electronic technology is such that factors affecting computation speed have almost been minimised; switching for instance is almost instantaneous. Electronic components are so good, in fact, that the time taken for a logic signal to travel between two points is now a significant factor of instruction times.

Clearly, with the actual physical size of components being very small and the high circuit density, there is little scope for improving computation speech significantly by such means as even denser circuitry or still faster electronic components. Thus, development of faster computers will require a new approach that depends on the imaginative use of existing knowledge.

One such approach is to increase computation speed through parallelism. Obviously, a parallel computer with p identical processors is potentially p times as fast as a single computer, although this limit can rarely be achieved.

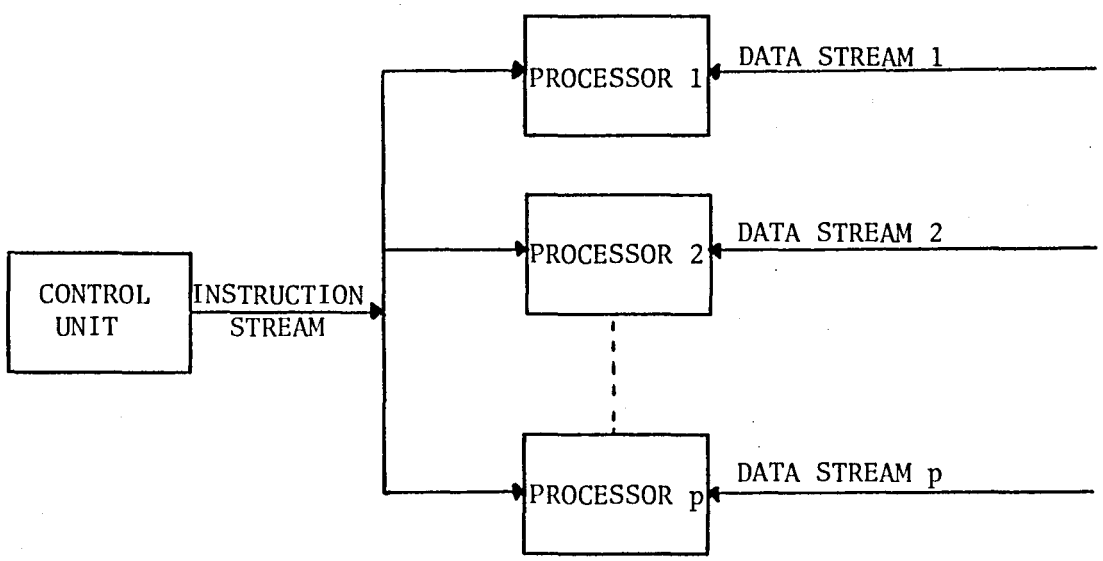
Parallelism has been developed in various forms and this has led to two general classes of parallel computers. These basic classifications made by Flynn [1966] are Single Instruction stream Multiple Data stream (SIMD) and Multiple Instruction stream Multiple Data stream (MIMD) computers. We shall discuss both of these types of parallel computer, outlining their differences and the advantages of each model. Another

type of computer, the pipeline computer, which is also sometimes classed as a parallel computer, will also be briefly described.

1.2 SIMD Computers

The SIMD parallel computer or Array processor is made up of an array of processors, each executing the same string of instructions on different data. A p processor SIMD computer is represented diagramatically in Figure 1.1.

Each of the processors in an SIMD computer differ from a standard computer in that they are unable to generate their own instructions. Instead, the instructions are provided by a control unit which is usually a computer itself. Associated with each processor is a private memory which provides it with its own data stream and consequently each processor executes the same instruction on its own data simultaneously. This leads to the definition of processors being synchronous when all instructions executed by the processors in parallel are identical.



SIMD COMPUTER

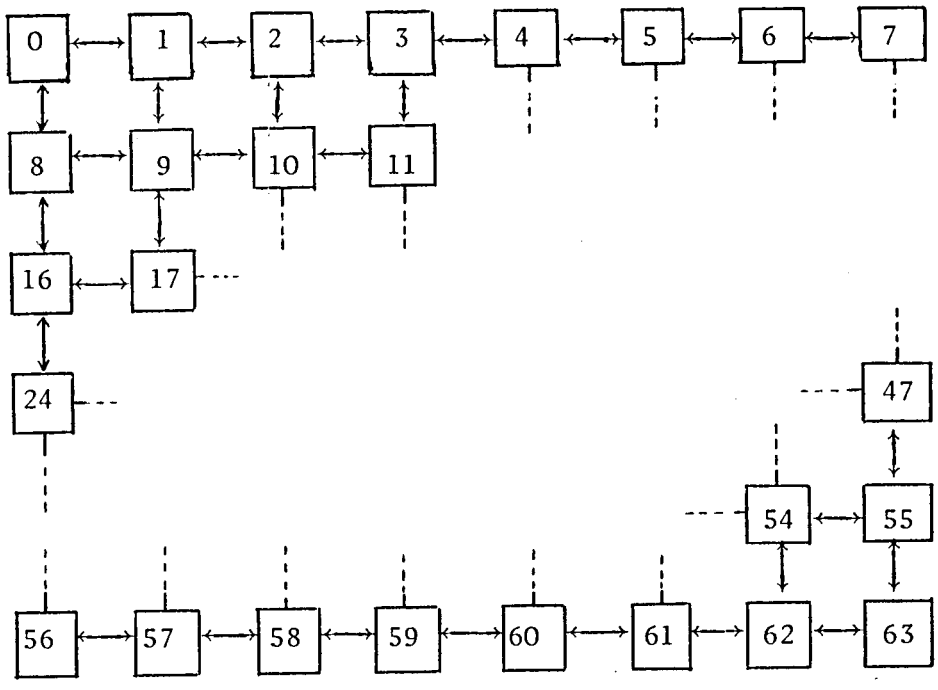
Figure 1.1

An example of an SIMD computer is the Illiac IV (Barnes et al [1968] and Bouknight et al [1972]), built by Burroughs Corporation and now located at the NASA Ames Research Centre, California. It comprises of 64 synchronous processors, each processor being almost a standard processor (by definition each processor lacks the ability to generate its own instructions). Obviously, expense severely limits the number of processors of this kind that may be combined to form an SIMD computer. However, SIMD computers under development employ large numbers of bit serial processors, e.g., the ICL Distributed Array Processor or DAP (Reddaway [1973]) which typically consists of 4096 microprocessors. Unfortunately, a bit serial processor is considerably slower than a standard computer (Parkinson, 1976) and the actual speed-up achieved by an SIMD computer using such processors is therefore that much less.

It is necessary for the processors in an SIMD computer to be able to communicate with one another. Unfortunately a complete interconnection network, where every processor is connected to every other processor, is expensive and unrealistic and so a reduced network of interconnections is necessary.

One such network is indicated in Figure 1.2 where the 64 processors form an 8×8 array, each processor being connected to its 4 immediate neighbours. This type of network is employed by both the Illiac IV and the DAP. From Figure 1.2 it is clear that this network is very suitable for the solution of partial differential equations in two dimensions which, typically, involves the application of an iterative formula of the form,

$$x_{i,j} = x_{i+1,j} + x_{i-1,j} + x_{i,j+1} + x_{i,j-1} - 4x_{i,j} . \qquad (1.2.1)$$



8×8 ARRAY PROCESSOR

Figure 1.2

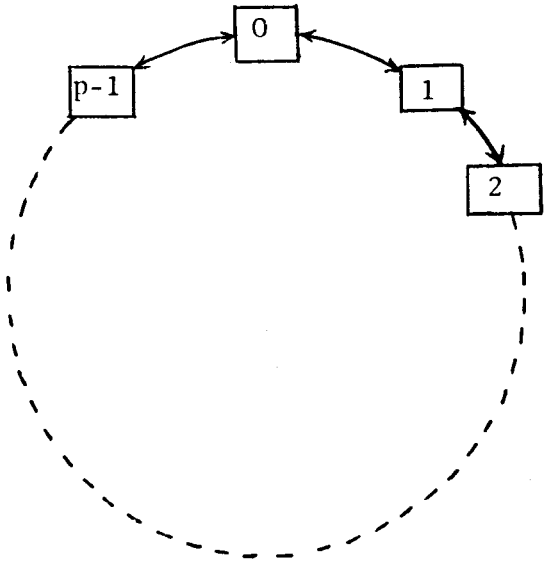
An alternative network is the cyclic interconnection network, illustrated in Figure 1.3, where the processors form a ring and again are connected to their immediate neighbours. This design is clearly suitable for algorithms containing assignment statements of the form,

$$x_i = x_{i-1} + x_{i+1} - 2x_i \quad . \tag{1.2.2}$$

Other interconnection networks do exist that are suitable for different types of algorithms. Unfortunately these networks are comparatively inflexible and when the requirements of a particular algorithm do not match the interconnection pattern of the computer, the communication delays incurred can seriously affect the execution time of the algorithm.

Another important feature that affects the class of problems for which the SIMD computer is suitable is its difficulty in dealing with conditional statements. A conditional statement can create more than one stream of instructions and since by definition there

can be only one stream of instructions, it is impossible to execute more than one of the branches of the conditional statement simultaneously. Each processor does however usually possess a local on/off switch or mask and so it is possible to prevent any of the processors from executing any of the instructions when necessary. Thus by setting the masks appropriately a conditional statement can be dealt with by executing each instruction stream that is created sequentially.



CYCLICALLY CONNECTED PROCESSORS

Figure 1.3

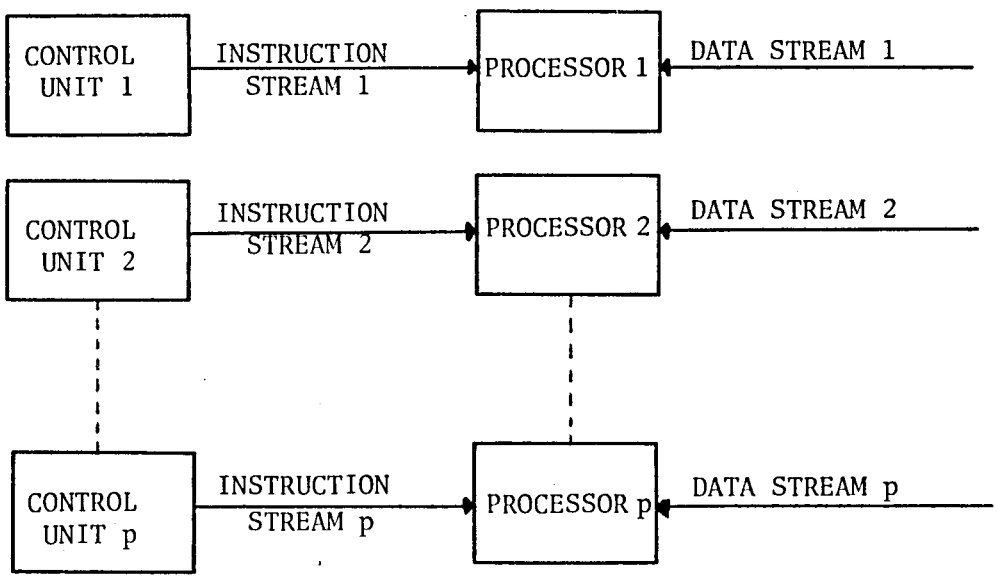
Clearly, the basic characteristics of the SIMD computer mean that the type of problem that may be solved efficiently on such a computer must have a high degree of parallelism so that as many of the available processors as possible can be used simultaneously. Also, a suitable interconnection network must be available to avoid excessive communication delays. Thus the SIMD computer is not a general purpose computer. However, there are a sufficient number of important problems,

mostly of a numerical nature (e.g. the solution of equations arising from the weather forecasting problem), suitable for SIMD computers to justify the development of special-purpose computers of this type.

1.3 MIMD COMPUTERS

The MIMD computer or multiprocessor is basically a minicomputer network. Each processor generates its own instruction stream which it executes on its own data stream. Such a computer with p processors is illustrated in Figure 1.4.

Each processor has its own control unit and so is able to generate its own instruction stream. Hence it is possible to execute different instructions simultaneously, which is our definition of asynchronous processors. Clearly, the independence of each processor means that they need not be identical, but they must be compatible with each other.



MIMD COMPUTER

Figure 1.4

Each processor also has its own data stream which is obtained from two sources. A large primary memory, usually referred to as the common memory, is accessible by each processor. Although the assumption is often made that each processor can obtain any piece of

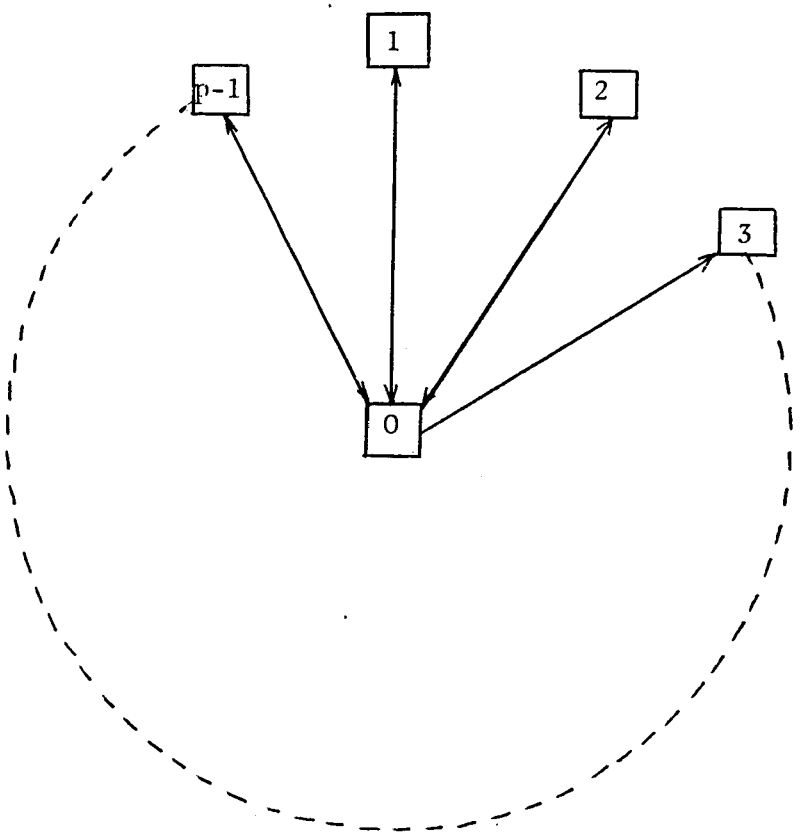
information from the common memory in unit time, in reality, there are complex problems involving such things as memory contention and processor interconnections. These complex problems can be reduced by the provision of a private memory associated with each processor in which important data is stored.

Examples of MIMD computers include the C.mmp multi-minicomputer (Wulf and Bell, 1972), under development at Carnegie-Mellon University, which is constructed of up to 16 asynchronous processors. Of particular interest is the Interdata Dual Processor Computer (Barlow et al, 1977) being developed at the Department of Computer Studies of Loughborough Univeristy which is considered in more detail in Section 1.5.

Obviously, the number of processors involved in existing MIMD computers is very small and the size of future computers will be restricted by expense. This number of processors is further restricted by the present unavoidable problems already mentioned, such as memory contention or store clashing, which grow exponentially with p, the increasing number of processors.

For MIMD computers with a very small number of processors it would be possible to implement a complete processor interconnection network. Otherwise a reduced network must be used such as those already mentioned for SIMD computers. Another interesting reduced network is the Star configuration, illustrated in Figure 1.5, where one processor is connected to each of the p-1 other processors.

The MIMD computer is clearly more flexible than the SIMD computer and so can be used to solve a greater variety of problems. The main difficulty that arises is the partitioning of the problem to yield an efficient method of solution rather than actually being able to solve the problem. Thus, the MIMD computer may be considered a general purpose computer.



STAR INTERCONNECTION NETWORK

Figure 1.5

1.4 PIPELINE COMPUTERS

Pipeline or Vector computers achieve an increase in computation speed by a novel approach to parallelism. This type of computer, although essentially sequential, achieves a form of parallelism by dividing arithmetic operations into subtasks and executing these subtasks on queues of pairs of operands simultaneously. Although pipeline computers are somewhat different to SIMD and MIMD computers, they are of interest because the form of algorithm that achieves a good speed up on a pipeline computer is closely linked with those best suited to SIMD computers.

Floating-point arithmetic operations may be considered as a

sequence of subtasks such as operand fetching, exponent adjustment, coefficient alignment etc. A pipeline computer separates these subtasks and by means of an instruction look-ahead mechanism sets up a queue of operand pairs on which to execute the operation. Then, in assembly line fashion, the queue of operand pairs provides a continuous stream of data for the sequence of subtasks. Each subtask acts on a pair of operands and then passes them to the next subtask while accepting the next pair of operands.

Examples of pipeline computers include the Control Data Corporation (CDC) STAR-100 (Hintz and Tate, 1972) and the Texas Instruments Advanced Scientific Computer (Watson, 1972).

In order to investigate how best to use pipeline computers we must examine the timings of the pipeline operations. The subtasks of an operation are designed so that each subtask is completed in a fixed amount of time τ or a cycle. We further define the total time to complete an operation as σ. Then the time required to perform n pipeline operations will be $(n-1)\tau+\sigma$, since the delay before the first result is produced will be σ after which further results are produced at the end of each cycle. Obviously, the time required to execute one instruction on a standard computer, say t, will be less than σ the time required by a pipeline computer. Thus to achieve a speedup when performing n operations we require

$$(n-1)\tau + \sigma < nt$$

$$\rightarrow \qquad n > \frac{(\sigma-\tau)}{(t-\tau)} \qquad (1.4.1)$$

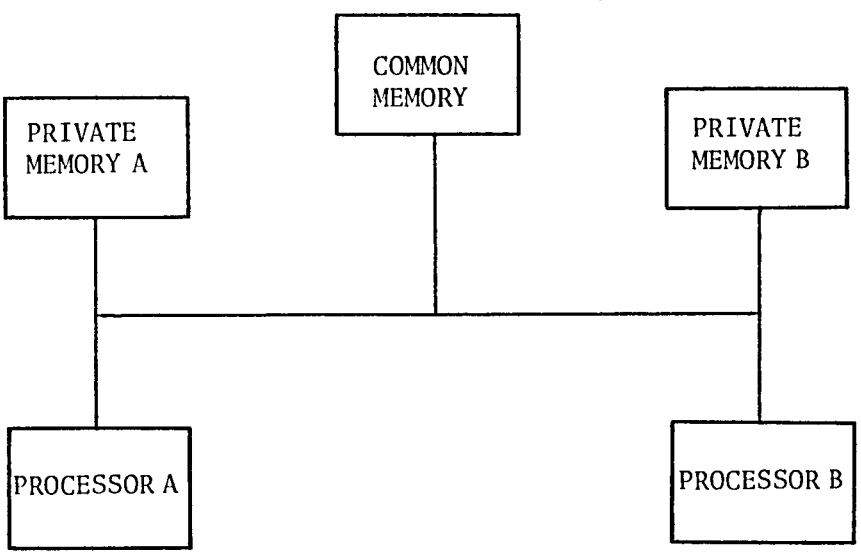
Clearly, to take full advantage of pipeline computers, algorithms must be designed so that this condition is often satisfied i.e., long sequences of identical operations are required as with the SIMD computer.

1.5 THE INTERDATA DUAL PROCESSOR

The type of parallel computer that this thesis is particularly concerned with is the MIMD computer and so in the final section of this chapter we shall examine in more detail the Interdata Dual Processor which, at present, is being developed at the Department of Computer Studies of Loughborough University.

The theoretical model of a dual processor is illustrated in Figure 1.6. The model consists of two processors A and B and associated with each processor is a private memory. In addition to this there is a common memory accessible by both processors but obviously not simultaneously. This model is essentially symmetric, in particular, with regard to accessing the common memory by either of the processors.

The actual configuration of the Interdata Dual Processor is illustrated in Figure 1.7. Although this is the present form of the computer, it was originally an Interdata model 55 dual communications processor (Interdata Inc., 1971). In the original form, processor B was an Interdata model 50 processor, the remainder of the system being the same as its present form.

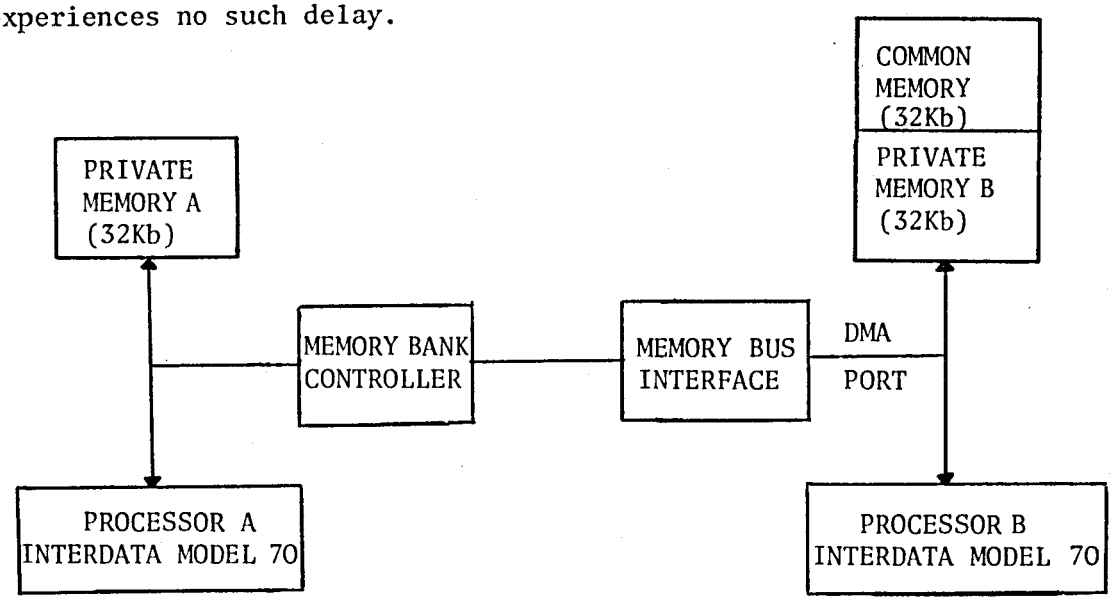


THEORETICAL MODEL OF DUAL PROCESSOR

Figure 1.6

The two processors A and B are identical Interdata model 70 processors. The model 70 is a 16 bit processor using 16 registers and working on an IBM 360-like instruction set. Instructions can be 16 or 32 bits long and take 1 or 2 μ seconds to load from memory. Floating point numbers are held as 32 bit fullwords while integers are held as 16 bit halfwords.

Processor A has 32K bytes of memory, called its private memory, which cannot be accessed by processor B. Processor B, however, has 64K bytes of memory, the first 32K bytes being the private memory of processor B which cannot be accessed by processor A. The second 32K bytes of processor B's memory is the common memory and can be accessed by both processors. The memory cycle time is 1 μ seconds. Processor A has direct access to the common memory via the memory bus interface. When accessing the common memory, processor A uses the actual physical address in the common memory and so the address translation function of the memory bank controller is not required. Hence, the only delay experienced by processor A when accessing common memory is ~ 1 μ second at the DMA (Direct Memory Access) port. Processor B, of course, experiences no such delay.



INTERDATA DUAL PROCESSOR CONFIGURATION

Figure 1.7

Memory contention or clashing occurs when both processors attempt to access common memory at the same time. A consequence of the asymmetry of the system is that if processor B is accessing its private or common memory, then processor A is locked out of the common memory until the memory cycle of B is completed. However if A is accessing common memory then B is locked out of both its private and common memory until the memory cycle of A is completed.

It appears that both processors are subject to a maximum delay of 1 μ second due to memory contention. However, due to the memory bus interface logic, processor A reserves the common memory 0.5 μ seconds before it requires it and so processor B is in fact subject to a maximum delay of 1.5 μ seconds due to memory contention. This also causes an overlap of the two delays that processor A is subject to and so the maximum additional delay that it can suffer due to memory contention is only 0.5 μ seconds.

The combined effect of both delays appears to be the same for both processors (1.5 μ seconds) but processor A in fact suffers more because it has a minimum delay of 1 μ second while processor B has a minimum delay of 0 μ seconds.

This completes the survey of current parallel computer architectures, and in the next chapter some basic techniques for developing algorithms suitable for parallel computers are introduced.

The computer architectures discussed so far have been based on the concept of 'control flow' or the stored program computer (Burke, Goldstine and Von Neumann, in Computer Structures, Bell and Newell, 1971) which imposes certain sequential restrictions which may be undesirable in parallel computers. An alternative approach is to base the design on the concept of 'data flow' in which the order of execution of instructions is dictated by the availability of data (Dennis and Misunas, 1974 and 1975).

The structure of a data-flow computer is essentially the same as that of an MIMD computer in that it is comprised of interconnected asynchronous processors each having access to its own private memory and a large shared memory (Rumbaugh, 1975).

CHAPTER 2

BASIC TECHNIQUES FOR PROGRAMMING A PARALLEL COMPUTER

2.1 INTRODUCTION

The existing form of standard computer algorithms, in particular, the classical methods of numerical analysis, are often unable to fully exploit the potential of the parallel computers described in Chapter 1. This clearly meant that a fresh look had to be taken at existing algorithms which has led to the reformulation of these algorithms or the development of new ones to give efficient parallel algorithms.

The development of parallel algorithms depends on the simple but extremely important observation that independent computations may be executed simultaneously. What is meant by independent computations? Computations may be described formally as independent if each result variable appears in only one computation, or in simple terms, if the results obtained from one computation are unaffected by the results obtained from another, then the two computations are independent.

As an example of independent computations, consider the addition of two n-vectors, i.e.,

$$\underline{c} = \underline{a} + \underline{b} \quad , \tag{2.1.1}$$

where

$$\underline{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} , \quad \underline{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \text{ and } \quad \underline{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} .$$

Obviously, the evaluation of the components of the result vector $\underline{c}$ are of the form,

$$c_i = a_i + b_i \qquad \text{for } i=1,2,\ldots,n \ , \tag{2.1.2}$$

and so the calculations are independent. A computer with n processors will clearly be able to calculate the result in one step. This example is also said to exhibit inherent parallelism; that is to say that it contains independent computations already, without the need of having to be reorganised.

A parallel algorithm may be created by the recognition of the inherent parallelism of a standard sequential algorithm, i.e., an algorithm designed for a single processor or sequential computer. When algorithms exhibit little or no inherent parallelism it is obviously necessary to reconstruct them so as to increase this property. This is often the case with good sequential algorithms since they have been designed specifically for sequential computers and what parallelism they do possess is usually obscured. For precisely this reason, good sequential algorithms do not always lead to good parallel ones.

When designing an algorithm for a parallel computer it is obviously necessary to take into consideration the basic characteristics of the computer. Now in Chapter 1, three different classes of parallel computers were described and so it is important to know if an algorithm designed for one type of parallel computer is a good algorithm for another type.

If we considered SIMD and MIMD computers first, we see that SIMD computers are usually larger than the MIMD type, i.e., SIMD computers possess up to $O(n^m)$, m=2,3,4, processors while existing MIMD computers have up to O(n) processors only, where n is the order of the problem. So to fully exploit the potential of an SIMD computer requires an algorithm with a higher degree of parallelism (a larger number of independent computations) than is necessary to exploit the potential of an MIMD computer. This does not mean that an algorithm designed for an MIMD computer cannot be run on an SIMD computer but that if it contains a maximum of n independent computations then only n of the processors of the SIMD computer may be used concurrently, the rest being superfluous. Conversely, an MIMD computer would have insufficient processors to execute $O(n^m)$ independent computations simultaneously but instead may execute them in groups of p computations if it has p processors.

In addition to this, the processors of an SIMD computer are synchronous and so are unable to take advantage of independent computations that are not identical but the processors of an MIMD computer are asynchronous and can take advantage of such computations. So clearly, non-identical computations must be executed sequentially on an SIMD computer which further reduces the number of its processors that may be used concurrently.

We further observe that, since the processors of an MIMD computer are asynchronous, they need not necessarily be involved on the same problem. Clearly then, if the addition of an extra processor has little or no effect on the run time of an algorithm it would be better to use that processor on a different problem. Thus, in the design of an algorithm for an MIMD computer we are interested in the efficient use of processors as well as the speed at which the problem can be solved.

The processors of an SIMD computer however do not possess this ability and when not required must therefore lie idle. So however small an improvement is achieved by the addition of an extra processor to execute an algorithm on an SIMD computer, if that extra processor is available it is better to use it. In the design of algorithms for SIMD computers we are therefore interested only in the speed in which a problem can be solved.

Clearly, the characteristics of the two classes of computers and the basic aims of interest when designing algorithms for them are such that a good MIMD algorithm is generally not a good SIMD algorithm and vice versa.

If we now consider pipeline computers, we see that a speed up is achieved by producing a string of identical operations that may be queued up and treated in assembly line fashion. It is not difficult to see that the string of operations must also be independent. Also,

the longer the string the greater the speed up achieved. Obviously then the requirements of a good pipeline algorithm are essentially the same as a good SIMD algorithm and so a good SIMD algorithm is usually a good pipeline algorithm.

Similar conclusions have also been reached by Stone [1973b], who goes as far as classifying pipeline computers as SIMD computers but modifies this statement by saying that results achieved by the study of array processors can generally but not always be applied to pipeline computers.

Once a parallel algorithm has been derived, it will of course be necessary to be able to assess its effectiveness. How much faster is the algorithm than the sequential algorithm or in fact other parallel algorithms? How efficient is it? Can it be improved on? These questions can be answered by use of the quantities T_p, S_p and E_p defined as follows:

> if T_p is the computation time for an algorithm run on a computer with p processors, in particular, T_1 is the sequential computation time (usually of the best sequential algorithm rather than the parallel algorithm that is being assessed), then the speed-up S_p, achieved by p processors is,
>
> $$S_p = T_1/T_p \quad , \tag{2.1.3}$$
>
> and the efficiency E_p is,
>
> $$E_p = S_p/p \quad . \tag{2.1.4}$$

It can be verified that these definitions are consistent with the uniprocessor case when p=1.

The majority of literature concerning parallel computers, in particular past surveys of parallel algorithms, including those of Miranker [1971], Poole and Voigt [1974] and Heller [1976], have been strongly orientated towards SIMD computers. This is because the problems associated with MIMD computers tend to be more difficult

to solve at present than those associated with SIMD computers. Accordingly, section 2.2 briefly describes this previous work and introduces some of the basic techniques involved in designing algorithms for SIMD computers. Sections 2.3 and 2.4 describe in more detail, similar aspects concerned with MIMD computers with a small number of processors.

It will be seen that there is a considerable difference between the design of algorithms for SIMD computers and MIMD computers. Since this thesis is concerned mainly with MIMD computers, in particular the Interdata Dual Processor, the remaining chapters investigate specific problems and develop parallel algorithms suitable for MIMD computers with a small number of processors.

2.2 THE DESIGN OF ALGORITHMS FOR SIMD AND PIPELINE COMPUTERS

In this section we consider the design of algorithms for both SIMD and pipeline computers since the approach is essentially the same. The previous surveys of algorithms for SIMD computers have already been mentioned and,in addition to these,similar work with respect to pipeline computers can be found in reports by Lambiotte [1975] and Lambiotte and Voigt [1975].

It has been shown that algorithms for SIMD computers require a high degree of parallelism, i.e., a large number of identical independent computations that can be executed simultaneously, and their aim is to reduce the number of steps to a minimum. Obviously, the addition of two n-vectors, described in equation (2.1.1) is ideal for an SIMD computer since it consists of n identical independent operations and may be computed in one step using n processors. If this is generalised to the addition of two (n×m) matrices, where an (n×m) matrix A is defined as,

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & & \vdots \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} \quad , \tag{2.2.1}$$

then clearly the addition may be performed in one step using n.m processors.

Consider now the matrix product,

$$C = A.B \quad , \tag{2.2.2}$$

where A is an (n×p) matrix, B is a (p×m) matrix and the (n×m) result matrix C is defined as

$$c_{ij} = \sum_{k=1}^{p} a_{ik}\, b_{kj} \quad , \quad \text{for } i=1,2,\ldots,n, \; j=1,2,\ldots,m. \tag{2.2.3}$$

The product consists of n.m identical independent computations and so each element of matrix C may be calculated simultaneously using n.m processors.

Obviously, vector and matrix operations are well suited to SIMD computers. Another powerful method for generating parallel algorithms is recursive doubling, so called because it divides the original computation into two independent smaller computations of equal complexity, which in turn are reduced to even smaller computations recursively. As an example, consider the sum of n numbers, $\sum_{i=1}^{n} a_i$, then clearly,

$$S_n = \sum_{i=1}^{n} a_i = \left(\sum_{i=1}^{m} a_i\right) + \left(\sum_{i=m+1}^{n} a_i\right) \text{ where } m=\lceil n/2 \rceil \tag{2.2.4}$$

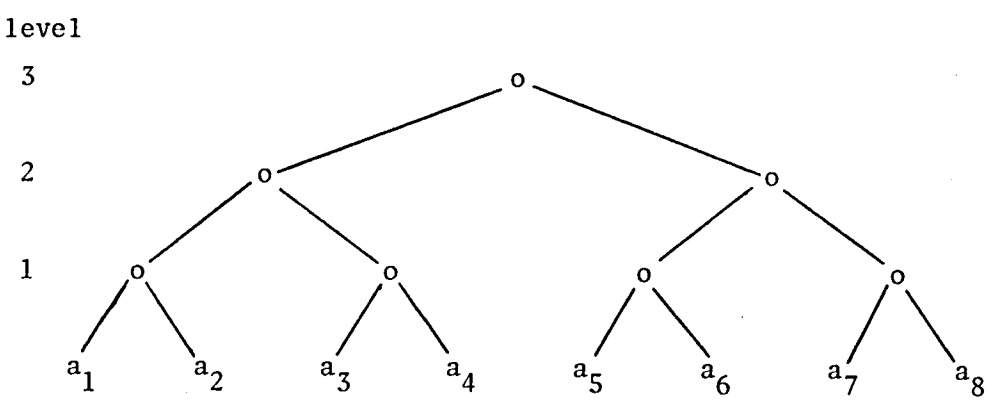
and repeated splitting leads to an algorithm that evaluates S_n in $\lceil \log_2 n \rceil$ steps using $\lceil n/2 \rceil$ processors, where $\lceil x \rceil$ is defined as the smallest integer greater than x.

This last example leads us to an optimum class of algorithms

(Heller, 1976) for evaluating expressions of the form,

$$A_n = a_1 \text{ o } a_2 \text{ o } a_3 \ldots\ldots \text{o } a_n \qquad (2.2.5)$$

where o is any associative operator. Applying recursive doubling to this expression produces an algorithm that is illustrated by the evaluation tree of Figure 2.1. At level 1 the operator o acts on adjacent pairs of operands, at level 2 it acts on adjacent pairs of results from level 1 and so on until the result A_n is produced. At each level the operations are independent and identical and so may be executed simultaneously. The first level has the greatest number of operations being $\lceil n/2 \rceil$, which means $\lceil n/2 \rceil$ processors will be sufficient to evaluate the operations at each level simultaneously. The number of levels is $\lceil \log_2 n \rceil$ and so by using $\lceil n/2 \rceil$ processors the result A_n may be evaluated in $\lceil \log_2 n \rceil$ steps. Heller named this algorithm the associative fan-in algorithm but it is more familiarly known as the log-sum and log-product when the operators are + and × respectively.

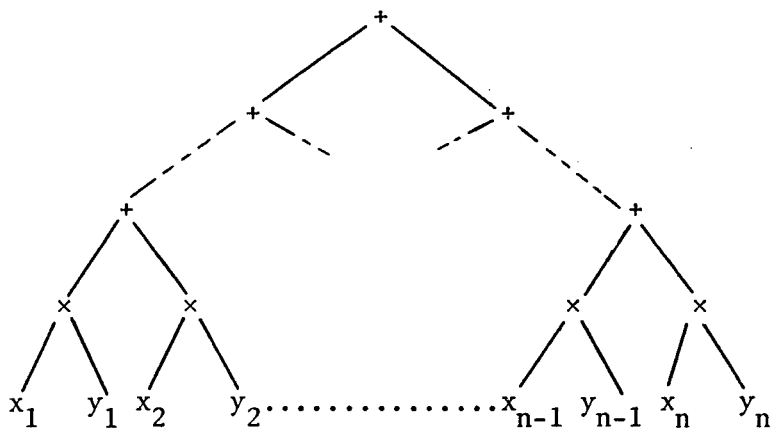


EVALUATION TREE

Figure 2.1

A special case of the associative fan-in algorithm is the inner or scalar product which has the form $\sum_{i=1}^{n} x_i y_i$ or the sum of the products $x_i y_i$ (i=1,2,...,n), and is illustrated in Figure 2.2. Obviously the n products may be performed simultaneously using n

processors followed by a log sum. Thus the result is produced in $\lceil \log_2 n \rceil + 1$ steps using n processors. The matrix product defined in (2.2.3) consists of n.m independent inner products and so clearly the matrix C may be evaluated in $\lceil \log_2 p \rceil + 1$ steps using n.m.p processors.



INNER PRODUCT

Figure 2.2

These are the basic computations that are used in the design of the majority of SIMD algorithms. Obviously, these forms of computations are also suited to pipeline computers and the algorithms based on them are therefore also suitable for pipeline computers.

The design of the parallel algorithm thus involves the restructuring of the sequential algorithm into a form that is usually a combination of these basic computations, e.g. the algorithm of Chen and Kuck [1975] for the solution of a triangular system of equations defined in Chapter 4 is basically a sequence of matrix sums and products.

In the development of a parallel algorithm it is often assumed that the computer has unlimited parallelism i.e. the computer has as many processors as are required. This often leads to an algorithm that requires an unrealistically large number of processors. A practical algorithm is then obtained by constructing a second algorithm that reduces the processor requirement to a realistic number without

significantly slowing the algorithm.

There are two basic principles by which the second algorithm is constructed, namely the algorithm decomposition and the problem decomposition principles (Hyafil and Kung, 1974). In the algorithm decomposition principle it is assumed that q_i operations are performed during step i of the original algorithm. If there is a maximum of p processors available, then $\lceil q_i/p \rceil$ steps are required to perform step i in the second algorithm. In the problem decomposition principle, the original problem of order n is partitioned into smaller problems of order p and the parallel algorithm is then applied to each of the smaller problems.

Numerous algorithms have been developed for SIMD computers using these basic techniques, most of which are included in the surveys by Miranker, Poole and Voigt and Heller. A typical problem that has been investigated is the solution of a linear system of equations (Pease [1967], Csanky [1975] and Sameh and Kuck [1975]). Specific forms of linear systems have also been investigated such as triangular systems (see Chapter 4), tridiagonal systems (Stone [1973a,1975a] and Heller, Stevenson and Traub [1974]) and block tridiagonal and banded systems (Heller [1974c] and Hyafil and Kung [1975]). Systems of equations arising from differential equations have been considered by Gilmore [1971], Liu [1974], Hayes [1974], and Sameh, Chen and Kuck [1974]. Other parallel algorithms that have been developed include parallel forms of Fast-Fourier transforms (Pease [1968], and Stone [1971]) and eigenvalue determination methods (Sameh [1971] and Sameh and Kuck [1971]). Various related problems have also been investigated in particular by Kogge, Stone, Kuck and Heller.

The algorithms are implemented on both SIMD and pipeline computers using vector instead of scalar operations. In the case of SIMD computers

special variables are defined that are dispersed throughout the private memories rather than special operators. When a special variable is used it refers to variables in the same position in each of the private memories rather than a single variable. Pipeline computers however define special vector operators that act on vector operands rather than single variables. For more specific information on programming and implementation on the Illiac IV we refer to Lawrie et al [1975] or Stevenson [1975] and for the CDC STAR-100 to Owens [1973].

2.3 CREATING MULTIPLE INSTRUCTION STREAMS ON AN MIMD COMPUTER

In order to create multiple instruction streams on an MIMD computer (i.e., implement parallel segments of an MIMD program), it is necessary to include additional statements in high level languages such as ALGOL and FORTRAN. This is because the processors of an MIMD computer function independently, and so must be able to let each other know when segments may be initialised and when they have been completed.

Obviously, it is necessary to be able to create and terminate parallel segments but it is also important to ensure that parallel computations are carried out correctly. As an example, suppose we wish to form the sum of the elements of the vector V[I] (I=1,2,...,N). To do this in parallel, each processor performs the statement,

SUM←SUM+V[I]

It is possible that the following sequence of operations may occur:

1. processor 1 fetches the value SUM from memory,
2. processor 2 fetches the value SUM from memory,
3. processor 1 adds $V[I_1]$ to its private value of SUM and restores the new value of SUM in memory,
4. processor 2 adds $V[I_2]$ to its private value of SUM and restores the new value of SUM in memory.

Clearly, the incorrect result is produced since the effect of adding $V[I_1]$ by processor 1 is lost. So it is necessary to safeguard against such an occurrence.

Various forms of statements have been investigated including the commands 'FØRK','JØIN','TERMINATE','ØBTAIN' and 'RELEASE' (in ALGOL 60 format) suggested by Anderson [1965] which are typical. The five statements have the following basic form:

```
        'FØRK' L1,L2;
LABEL:'JØIN' L1,L2,.....LN;
LABEL:'TERMINATE' L1,L2,.....LN;
        'ØBTAIN' V1,V2,....VN;
and     'RELEASE' V1,V2,.......VN;
```

where LI represents a label and VI represents a variable. We shall consider each statement in turn, giving a description of their purposes.

The fork statement - initialises two instruction streams, one starting at the statement labelled L1 and the other at the statement labelled L2. In Algol there are certain restrictions on the use of labels which also apply to this statement and so L1 and L2 must be local labels.

The join statement - terminates the parallel paths (instruction streams) in which it occurs. Each parallel path ends with a 'GØ TØ' statement to a labelled join statement. The label list included in the 'JØIN' statement contains the labels of the first statements of each of the paths that it terminates. The statement immediately following the join statement is not executed until all the paths contained in the label list have been terminated.

The terminate statement - is used to explicitly discontinue program paths. The fork statement dynamically activates program paths and the terminate statement is used to avoid creating a backlog of meaningless incomplete activations.

The obtain statement - provides exclusive use of the variables contained in the variable list. It is used to 'lock out' other parallel program paths from the use of those variables so as to avoid mutual interference.

The release statement - is the logical counterpart of the obtain statement. It allows access to variables (contained in variable list) that have been previously locked out by an obtain statement. Since it only releases those variables in the variable list, it may be applied selectively.

The actual implementation of these commands will be dependent on the characteristics of the parallel computer but they do have a general form. The execution of a fork statement creates two parallel program paths, one of which (usually the first one) is carried out by the processor that executes the fork statement. The other path is assigned to an available processor but in the event of no processor being available it is placed in a queue until one does become available.

The join and terminate statements control counters initialised to the number of labels in their label lists. Each time the statement is executed the corresponding counter is decreased by one and compared to zero. When the counter is not zero, the path is terminated, the processor that executed the path is released and if there are paths waiting to be executed, it is assigned to the path at the head of the queue. If the counter is zero, the path is terminated and the processor proceeds to the next program segment starting at the statement immediately after the join statement.

The obtain and release statements are more difficult to implement and depend on the hardware capabilities of the computer. It is interesting however to consider what happens when a processor requests a piece of data that has been restricted to the exclusive use of another

processor by an obtain statement. The path being executed by the processor can be suspended awaiting access and the processor reassigned to other work or the processor can be held in a state of idleness, continually trying to access the data until it is released by a release statement.

These commands are typical of those used by MIMD computers and now as a specific example we shall consider the commands used by the Interdata Dual Processor.

The programming language available on the Interdata Dual Processor is Fortran and the set of additional commands necessary to create parallel program segments include $FØRK,$JØIN,$DØPAR and $PAREND, plus two subroutines GETRES and PUTRES (Barlow et al , 1977). The four commands are macros that are expanded by the Fortran Macro Processor to Fortran code acceptable to the compiler.

Let us first consider the two commands $FØRK and $JØIN, which always occur in pairs as follows:

```
      $FØRK L1,L2,....LN;L
L1    .                    }
      .                    }Program segment 1
      .                    }
      GØ TØ L
L2    .                    }
      .                    }Program segment 2
      .                    }
      GØ TØ L                    :
      .                          :
      .                          :
      .                          :
LN    .                    }     :
      .                    }Program segment N
      .                    }
      .
      GØ TØ L
L     $JØIN
```

The $FØRK statement creates an arbitrary number of parallel paths each starting at the statements whose labels appear in the label list of the $FØRK statement and ending with a go to L, the label of the corresponding $JØIN statement. The labels in the label list are separated by commas,

the last one being followed by a semi-colon and the label of the corresponding $JØIN statement.

The two commands $DØPAR (or $DØPARALLEL) and $PAREND (or $PARALLELEND) are essentially a parallel form of DØ loop and are used as follows:

```
      $DØPAR 1 I=N1,N2,N3
         :                  } Program segment
         :
    1 $PAREND
```

where the control variable I (as in the DØ statement) set initially to N1, is incremented by N3 until greater than N2. The $DØPAR command creates one program path for each value of the control variable, but instead of each path being executed sequentially they are executed concurrently.

The $DØPAR and $PAREND are used to replace the DØ loop when the computations involved in each execution of the loop are independent (which means they may be executed in parallel). Obviously, to use the $FØRK and $JØIN statements to perform each loop in parallel would mean that the instructions included in the loop would have to be repeated for each value of the control variable. This is of course unnecessary with $DØPAR which is essentially an extension of the $FØRK instruction and so should be used.

Both pairs of commands are implemented in the same way. An entry, containing necessary information, is placed in a queue for each program path created by the $FØRK or $DØPAR instruction. The processor that executed the $FØRK or $DØPAR instruction then takes the first path from the queue and executes it, followed by the other available processors. The instructions $JØIN and $PAREND are counters which are set initially to the number of parallel program paths and decremented each time a program path is completed. On completion of a path, the processor that executed it is reassigned to the next path in the queue. One peculiarity of the Interdata Dual Processor is that the

statements following the $JØIN or $PAREND statement must be executed by the processor that executed the $FØRK or $DØPAR statement.

The subroutines GETRES and PUTRES are similar to the ØBTAIN and RELEASE commands in that they also prevent mutual interference between processors. Instead of giving exclusive use of certain variables to one processor, they give it exclusive use of a segment of program that contains these variables. The subroutines are implemented by creating an abstract resource ring that consists of abstract resources available to all processors. A resource may be possessed by only one processor at a time, other processors requiring it having to wait until it is given to them by the processor that possesses it.

A segment of program that we wish to give exclusively to one processor is made into resource I by placing it between two subroutines thus:

```
CALL GETRES(I)
  .
  .          } Program segment
  .
CALL PUTRES(I)
```

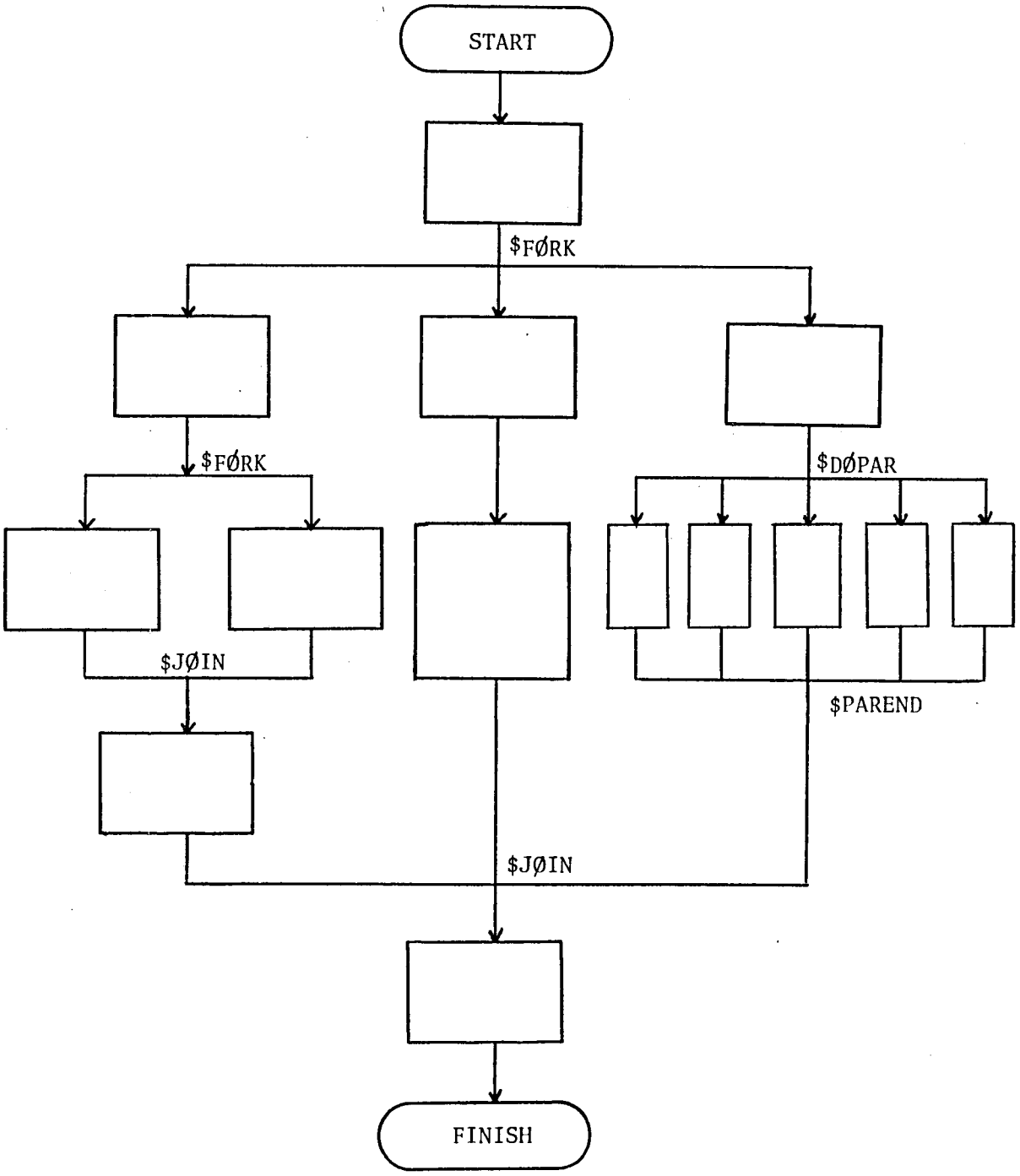
The segment only becomes available to other processors when the PUTRES subroutine call has been made.

The flowchart in Figure 2.3 illustrates the form that a program for an MIMD computer might take.

The general rules for the order in which the program segments are executed are quite simple. A segment of program that appears before a $FØRK or $DØPAR statement must be executed before that $FØRK or $DØPAR is executed. The program paths created by a $FØRK or $DØPAR statement can be executed simultaneously but if there are insufficient processors to execute all of the paths, the order in which they are executed is not important. The program segment following a $JØIN or $PAREND statement can only be executed when all the paths entering that $JØIN

or $PAREND statement have been completed.

Finally we see from Figure 2.3 that nesting of $JØINs and $DØPARs is permitted.



FLOWCHART STRUCTURE OF AN MIMD PROGRAM

Figure 2.3

2.4 THE DESIGN OF ALGORITHMS FOR MIMD COMPUTERS

In this section we shall investigate the inherent parallelism of existing algorithms, in particular, for MIMD computers with two processors but with a view to extending the ideas to computers with more processors. At first we shall consider some simple expressions and then progress to some specific algorithms.

Let us first consider expressions of the form of equation (2.2.5), i.e.,

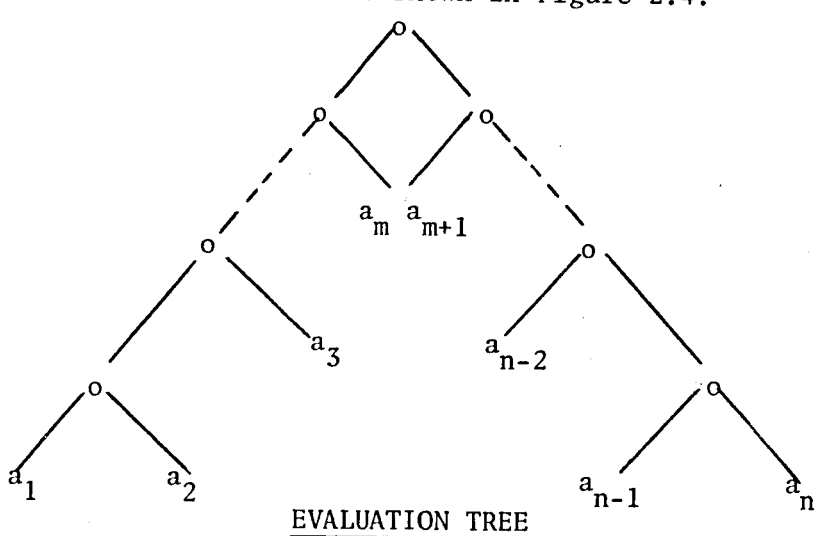
$$A_n = a_1 \circ a_2 \circ \ldots \circ a_n$$

which is evaluated using the associative fan-in algorithm on an SIMD computer. Observation of the evaluation tree corresponding to this algorithm (Figure 2.1) reveals that although operations at the same level in the tree are independent, those at different levels are not. Since the processors of an MIMD computer are asynchronous it would be preferable therefore to remove as much of the dependency as possible. This may be achieved by partitioning the problem once thus,

$$A_n = (a_1 \circ a_2 \circ \ldots a_m) \circ (a_{m+1} \circ a_{m+2} \circ \ldots . a_n) \qquad (2.4.1)$$

where $$m = \begin{cases} \frac{n}{2} \text{ when n is even} \\ (n+1)/2 \text{ when n is odd,} \end{cases}$$

which yields the evaluation tree shown in Figure 2.4.



EVALUATION TREE

Figure 2.4

Clearly the evaluation of each branch is independent and so may be done concurrently using 2 processors. Using the fork and join statements this may be programmed easily as follows:

```
      'FØRK' L1,L2;
L1:A1←A[1];
   'FØR' I←2 'STEP' 1 'UNTIL' M 'DØ' A1←A1oA[I];
   'GØTØ' L3;
L2:A2←A[M+1];
   'FØR' I←M+2 'STEP' 1 'UNTIL' N 'DØ' A2←A2oA[I];
   'GØTØ' L3;
L3:'JØIN' L1,L2;
   AN←A1oA2;
```

Obviously, for this expression we have,

$$T_1 = (n-1) \text{ operations}$$

and

$$T_2 = \begin{cases} \frac{n}{2} \text{ operations}, & \text{for n even} \\ (n+1)/2 \text{ operations}, & \text{for n odd}, \end{cases}$$

remembering that when n is odd one branch of the evaluation tree has one more operation than the other. Thus, the speed-up and efficiency are

$$S_2 = \begin{cases} 2(n-1)/n = 2-2/n, & \text{for n even} \\ 2(n-1)/(n+1) = 2-4/(n+1), & \text{for n odd}, \end{cases}$$

and

$$E_2 = \begin{cases} 1-1/n, & \text{for n even} \\ 1-2/(n+1), & \text{for n odd}, \end{cases}$$

which are almost optimum results.

If this strategy is now applied to the inner or scalar product we have

$$\text{S.P.} = \sum_{i=1}^{m} x_i y_i + \sum_{i=m+1}^{n} x_i y_i \quad , \tag{2.4.2}$$

where m is as defined for equation (2.4.1).

The corresponding evaluation tree and program will have the same form as those for the expression (2.4.1). Again, it is obvious that for the scalar product,

$$T_1 = n(M+A)-A$$

and

$$T_2 = \begin{cases} n(M+A)/2, & \text{for n even} \\ (n+1)(M+A)/2, & \text{for n odd,} \end{cases}$$

where M and A are the times required to perform a multiplication and an addition respectively. This leads to the results,

$$S_2 = \begin{cases} 2-2A/(M+A)n\ , & \text{for n even} \\ 2-2/(n+1)-2A/(M+A)(n+1), & \text{for n odd} \end{cases}$$

and

$$E_2 = \begin{cases} 1-A/(M+A)n\ , & \text{for n even} \\ 1-1/(n+1)-A/(M+A)(n+1), & \text{for n odd.} \end{cases}$$

Once again these results are very close to the optimum values.

Another expression that may be evaluated in a similar fashion is the polynomial of the form:

$$p = a_0 + a_1x + \ldots\ldots a_nx^n \quad . \tag{2.4.3}$$

The sequential computation time of a polynomial is uniquely minimised/assuming no preprocessing of coefficients by applying Horner's Rule (Borodin, 1971), which expresses the polynomial in the form:

$$p = (\ldots((a_nx+a_{n-1})x+a_{n-2})x\ldots a_1)x+a_0 \quad . \tag{2.4.4}$$

The partitioning of Horner's rule suggested by Dorn [1962] expresses the polynomial in the required form thus,

$$\left.\begin{aligned} p_1 &= (\ldots((a_\ell x^2+a_{\ell-2})x^2\ldots+a_2)x^2+a_0 \\ p_2 &= (\ldots((a_k x^2+a_{k-2})x^2\ldots+a_3)x^2+a_1 \\ \text{and}\quad p &= p_1 + p_2x \end{aligned}\right\} \tag{2.4.5}$$

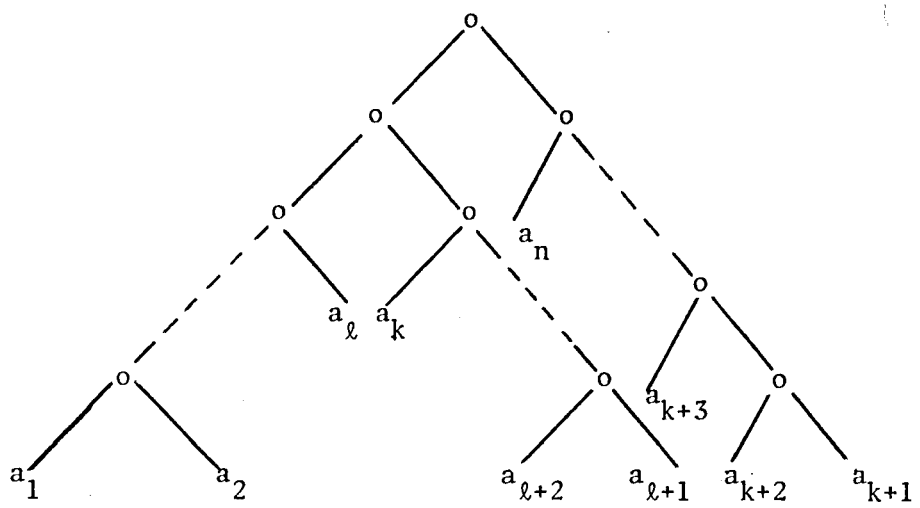
where ℓ=n and k=(n-1) for n even and ℓ=(n-1) and k=n for n odd. The speed-up and efficiency achieved by this method of evaluation are also impressive.

Clearly a similar strategy may be applied to these expressions for evaluating them on a p-processor computer. We simply partition the expression into p smaller expressions of equal size. As an

example, consider the evaluation of expression (2.2.5) using 3 processors, for which we partition the expression thus,

$$A_n = (a_1 o a_2 o \ldots a_\ell) o (a_{\ell+1} o \ldots a_k) o (a_{k+1} o \ldots a_n) \qquad (2.4.6)$$

where $\ell = \lceil n/3 \rceil$ and $k = \lceil 2n/3 \rceil$. The evaluation tree is given in Figure 2.5 and the program will be of the same form as that for a 2 processor computer but with 3 paths created by the fork statement.



EVALUATION TREE

Figure 2.5

Assuming of course that n is exactly divisible by 3, we then have,

$$T_3 = n/3+1 \qquad \text{operations ,}$$

$$S_3 = 3-12/(n+3) \ ,$$

and

$$E_3 = 1-4/(n+3) \ ,$$

which again is impressive but not quite as good as the speed-up and efficiency achieved when using 2 processors. It is not difficult to see that for a p processor computer we have,

$$T_p = n/p - 1 + \lceil \log_2 p \rceil \qquad \text{operations ,}$$

assuming n is divisible by p, and

$$S_p = p - \frac{(p+p^2(\lceil \log_2 p \rceil - 1))}{(n+p(\lceil \log_2 p \rceil - 1))}$$

and

$$E_p = 1 - \frac{(1+p(\lceil \log_2 p \rceil - 1))}{(n+p(\lceil \log_2 p \rceil - 1))} \quad .$$

As expected, the efficiency of the algorithm decreases as p, the number of processors, is increased.

Now let us consider the simple matrix operations, addition, subtraction and multiplication. First, we have the addition of two (n×m) matrices such as,

$$C = A + B \quad , \tag{2.4.7}$$

which is defined as

$$c_{ij} = a_{ij} + b_{ij} \qquad \text{for } i=1,2,\ldots,n, \quad j=1,2,\ldots,m.$$

When considering SIMD computers we established that the evaluation of C is made up of n.m independent operations and so the problem is simply to divide these operations equally between the two processors.

It is obvious that there are numerous ways of dividing the operations into two equal parts. If either or both of n and m are even, we may simply evaluate the odd numbered columns (or rows) of C using one processor and the even numbered ones using the other processor. The following program evaluates C by assigning alternate columns to the two processors,

```
      'FØRK' L1,L2;
L1:'FØR' J←1 'STEP' 2 'UNTIL' M 'DØ'
   'FØR' I←1 'STEP' 1 'UNTIL' N 'DØ' C[I,J]←A[I,J]+B[I,J];
   'GØTØ' L3;
L2:'FØR' J←2 'STEP' 2 'UNTIL' M 'DØ'
   'FØR' I←1 'STEP' 1 'UNTIL' N 'DØ' C[I,J]←A[I,J]+B[I,J];
   'GØTØ' L3;
L3:'JØIN' L1,L2;
```

It is not difficult to see that if either or both of n and m are even then,

$$S_2 = 2 \qquad \text{and} \qquad E_2 = 1 \ .$$

These results appear to be perfect but unfortunately an overhead is incurred by the use of the fork and join statements and so in fact $S_2<2$ and $E_2<1$.

Since matrix subtraction and multiplication also consist of the evaluation of the n.m elements of the result matrix, each of which are independent, exactly the same strategies may be applied, achieving identical speed-ups and efficiencies. Note however, from the matrix product defined in equation (2.2.3), that each element of the result matrix is a scalar product. So an alternative method of evaluating the matrix product is to calculate the elements of the result matrix one at a time using the scalar product algorithm for two processors already defined. It is a trivial problem for these algorithms to be extended so as to be suitable for a p processor computer.

At this point we shall consider an important difference between the SIMD and MIMD types of parallel computer. It is obvious that the processors of an SIMD computer are synchronized as well as synchronous i.e., as well as each processor executing the same instructions, the instructions are executed at exactly the same time. Not so obvious is the fact that even if the instruction streams of an MIMD computer are identical, the processors may not execute each instruction at exactly the same time. It is reasonable to assume that the processors of an MIMD computer are identical. The delays that they are subject to due to memory contention are not however the same and so even if the processors are initially synchronized, they will not usually be so for long.

Arising from this we see that although the previously described algorithms divide the total work into equal quantities, we can not be

sure that each processor finishes its work at the same time. To overcome this problem on a 2 processor computer, Kung [1976] suggests the use of a deque (double ended queue). If all the operations in any one of the algorithms already presented in this section (except the polynomial evaluation) are placed in a queue, then we may permit each processor to take operations from opposite ends of the queue. Although the processors may not meet exactly at the middle of the queue, clearly, the important point is that both processors will be kept occupied. Obviously, it would not be easy to use a deque for more than 2 processors.

In the remainder of this section we shall investigate algorithms for the solution of a system of linear equations of the form

$$A\underline{x} = \underline{d} \tag{2.4.8}$$

where A is an (n×n) matrix and $\underline{x}$, the solution vector, and $\underline{d}$ are (n×1) vectors. There are two classes of direct methods for the solution of such systems of equations, namely elimination methods and factorisation methods. The elimination method most commonly used is the Gauss Elimination Algorithm (Wilkinson, 1965) which transforms matrix A into an upper triangular matrix and, by a backward substitution process, computes the solution vector $\underline{x}$. If the original system (2.4.8) is denoted as

$$A^{(1)}\underline{x} = \underline{d}^{(1)} \quad , \tag{2.4.9}$$

then A is triangularised by the production of the sequence of systems,

$$A^{(r)}\underline{x} = \underline{d}^{(r)} \quad \text{for } r=2,3,\ldots,n \tag{2.4.10}$$

where $A^{(n)}$ is the required upper triangular matrix. At the r^{th} step of the algorithm, $A^{(r)}$ has the form,

$$A^{(r)} = \begin{bmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1n} \\ & a_{22} & & & \vdots \\ & & \ddots & & \\ & & & a_{rr} & \cdots & a_{rn} \\ & 0 & & \vdots & & \vdots \\ & & & a_{nr} & \cdots & a_{nn} \end{bmatrix}$$

and $A^{(r+1)}$ is derived from $A^{(r)}$ by subtracting a multiple m_{ir} of the r^{th} row from the i^{th} row for i=r+1,....,n. The same operations are performed on the right hand side vector $\underline{d}^{(r)}$ to produce $d^{(r+1)}$. The multipliers m_{ir}, chosen so as to eliminate a_{ir} (i=r+1,...,n), are defined as

$$m_{ir} = a_{ir}^{(r)}/a_{rr}^{(r)} \quad \text{for } i=r+1,\ldots,n. \tag{2.4.11}$$

Obviously, the first r rows of $A^{(r)}$ and $d^{(r)}$ will be unaltered and since the zeros in the first (r-1) columns are only replaced by a linear combination of zeros, they too will be unaltered. The remaining elements of $A^{(r+1)}$ and $d^{(r+1)}$ are defined by the following equations:

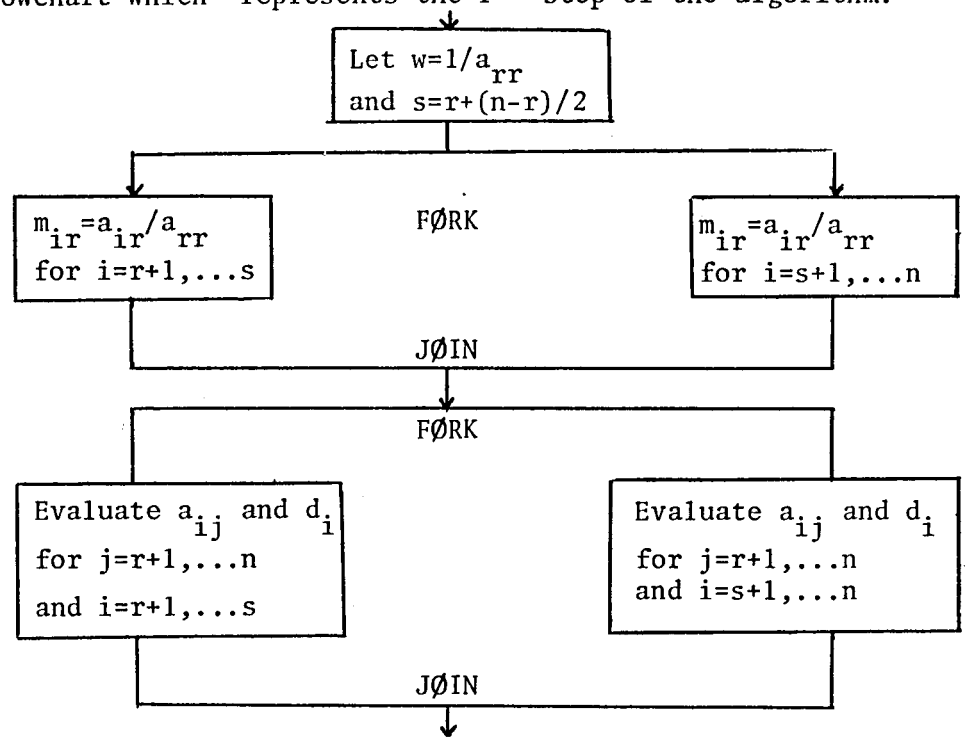
$$\left.\begin{aligned} a_{ij}^{(r+1)} &= a_{ij}^{(r)} - m_{ir}a_{rj}^{(r+1)} \quad \text{for } j=r,r+1,\ldots n \\ \text{and} \qquad d_i^{(r+1)} &= d_i^{(r)} - m_{ir}d_r^{(r+1)} \end{aligned}\right\} \text{for } i=r+1,\ldots n \tag{2.4.12}$$

The backward substitution process for the evaluation of $\underline{x}$ is then defined as,

$$x_i = (d_i - \sum_{j=i+1}^{n} a_{ij}x_j)/a_{ii} \quad , \quad \text{for } i=n,n-1,\ldots 1. \tag{2.4.13}$$

If we consider one step of the triangularisation process, we see that it requires the evaluation of (n-r) multipliers and (n-r)(n-r+1) elements of $A^{(r+1)}$ and $d^{(r+1)}$. Clearly the multipliers may be evaluated simultaneously and so may the elements of $A^{(r+1)}$ and $d^{(r+1)}$. Thus the process can be implemented on an MIMD computer with p processors by dividing each set of calculations into p equal groups. This may typically be done on a two processor computer as shown in the following

flowchart which represents the r^{th} step of the algorithm.



Clearly the speed-up for the triangularisation process achieved by using two processors in this way is,

$$S_2 \simeq \frac{2[6D+2n(n+1)S+n(2n+5)M]}{[12D+(2n^2+2n+3)S+(2n^2+5n+6)M]} < 2 \quad , \tag{2.4.14}$$

where D,S and M are the times required to perform a division, a subtraction and a multiplication respectively, and the efficiency is,

$$E_2 = \frac{S_2}{2} < 1 \quad . \tag{2.4.15}$$

The backward substitution process is essentially sequential, but methods for the parallel solution of triangular sets of equations are described in Chapter 4. An alternative approach is to take advantage of the fact that each element of $\underline{x}$ is a scalar product. Applying the methods already developed for scalar products, we obtain the following results for the execution of the backward substitution process on a 2 processor computer,

$$\left.\begin{aligned} S_2 &\simeq \frac{2[2D+(n-1)S+(n-1)M]}{[4D+(n+2)S+(n+2)M)]} < 2 \\ \text{and} \quad E_2 &= \frac{S_2}{2} < 1 \end{aligned}\right\} \quad . \tag{2.4.16}$$

Another elimination method is the Gauss Jordan Elimination Algorithm which reduces the matrix A to a diagonal matrix of the form,

$$A^{(n)} = \begin{bmatrix} a_{11}^{(n)} & & & 0 \\ & a_{22}^{(n)} & & \\ & & \ddots & \\ 0 & & & a_{nn}^{(n)} \end{bmatrix} .$$

The diagonalisation procedure is the same as the triangularisation procedure of Gauss Elimination except that during the r^{th} step, the (n-1) multipliers m_{ir} defined as

$$m_{ir} = a_{ir}^{(r)}/a_{rr}^{(r)} \qquad \text{for } \substack{i=1,2,\ldots n \\ i\neq r}$$

are chosen so to eliminate the r^{th} column of $A^{(r)}$ except $a_{rr}^{(r)}$.

The speed-up and efficiency for Gauss-Jordan Elimination are thus,

$$\left.\begin{aligned} S_2 &\simeq \frac{2[2D+(n-1)(n+1)S+(n-1)(n+3)M]}{[4D+(n-1)(n+1)S+(n-1)(n+3)M]} < 2 \\ \text{and} \qquad E_2 &= \frac{S_2}{2} < 1 \end{aligned}\right\} \qquad (2.4.17)$$

The solution vector $\underline{x}$ is then defined by the equations

$$x_i = d_i/a_{ii} \qquad i=1,2,\ldots,n \ , \qquad (2.4.18)$$

which may obviously be divided equally between two processors.

Arising from these two algorithms is a method for the evaluation of the determinant of a matrix. If A is reduced to either of the forms produced by the elimination algorithms, then the determinant of A is defined as

$$\det A = \prod_{i=1}^{n} a_{ii} \ , \qquad (2.4.19)$$

which is of the same form as equation (2.4.1) and so can be evaluated in the same way.

The second class of methods for the solution of (2.4.8) is

factorisation methods which are typified by the LU factorisation algorithm. This algorithm factorises A into two matrices L and U such that,

$$A = L.U \quad , \tag{2.4.20}$$

where,

$$L = \begin{bmatrix} 1 & & & \\ \ell_{21} & 1 & & \\ \ell_{31} & \ell_{32} & 1 & \\ \vdots & \vdots & & \ddots \\ \ell_{n1} & \ell_{n2} & \cdots & 1 \end{bmatrix} \text{ and } U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & & u_{2n} \\ & & \ddots & \vdots \\ & & & u_{nn} \end{bmatrix} .$$

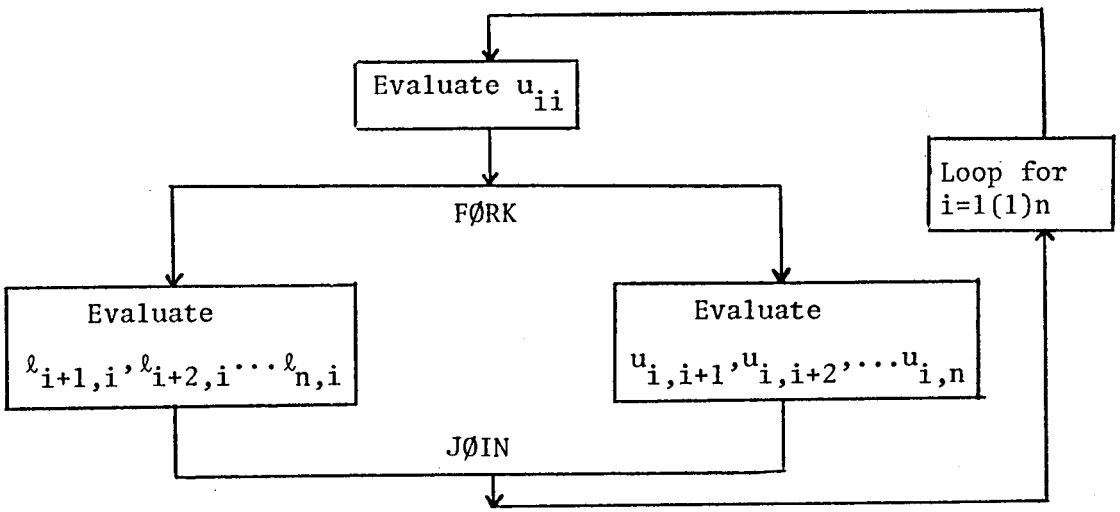
Then by introducing an auxiliary vector $\underline{y}$ such that $\underline{y}=U\underline{x}$, the solution vector $\underline{x}$ may be evaluated by performing forward and backward substitution processes respectively on the two triangular systems of equations,

$$L\underline{y} = \underline{d} \quad \text{and} \quad U\underline{x} = \underline{y} \ .$$

The elements of the matrices L and U may be found by forming the product LU and equating it to A to give the following formulae,

$$\left.\begin{aligned} u_{ij} &= a_{ij} - \sum_{k=1}^{i-1} \ell_{ik}u_{kj} \qquad \text{for } j= i,i+1,\ldots n \\ \text{and} \quad \ell_{ji} &= (a_{ji} - \sum_{k=1}^{i-1} \ell_{jk}u_{ki})/u_{ii} \quad \text{for } j=i+1,\ldots n \end{aligned}\right\} \text{for } i=1,2,\ldots n. \tag{2.4.21}$$

Clearly the evaluation of the elements in the i^{th} row of U and i^{th} column of L, apart from u_{ii}, are independent and so may be done simultaneously. The order in which the elements can be evaluated on a two processor computer is illustrated in the following flowchart:



If the factorisation is performed in this manner, then the speed-up and efficiency achieved are,

$$\left.\begin{aligned} S_2 &= \frac{[3D+(2n-1)S+(2n-1)M]}{[3D+(n+1)S+(n+1)M]} < 2 \\ \text{and} \quad E_2 &= \frac{S_2}{2} < 1 \quad . \end{aligned}\right\} \qquad (2.4.22)$$

The two substitution phases are defined as,

$$\left.\begin{aligned} y_i &= d_i - \sum_{k=1}^{i-1} \ell_{ik} y_k \quad \text{for } i=1,2,\ldots n \\ \text{and} \quad x_i &= (y_i - \sum_{k=i+1}^{n} u_{ik} x_k)/u_{ii} \quad \text{for } i=n,n-1,\ldots 1 \end{aligned}\right\} \qquad (2.4.23)$$

which may be treated in the same way as the substitution process of Gauss Elimination, except that the solution of $L\underline{y} = \underline{d}$ does not require any divisions.

A problem associated with solution of linear systems of equations is matrix inversion, which involves the solution of the matrix equation

$$AX = I \quad , \qquad (2.4.24)$$

where the unknown matrix X is the inverse of A and I (the identity matrix) is a unit diagonal matrix. Clearly, if A is an (n×n) matrix, then the problem involves the solution of n systems of equations of the form of (2.4.8), each system having the same left hand side but different right hand sides. Thus to compute the inverse of A requires one application of, for example, an elimination procedure followed by n independent substitution processes. The substitution processes may

be divided equally between the p processors, each substitution process being executed sequentially. Obviously if n is exactly divisible by p, the speed-up and efficiency of the substitution process when executed on a p processor computer will be,

$$S_p = p \quad \text{and} \quad E_p = 1 \ .$$

When matrix A is sparse i.e., many of the elements of A are zero, the algorithms already described become inefficient due to redundant operations (e.g. the elimination of elements that are already zeros), Special algorithms therefore exist for the solution of the system of equations (2.4.8) when matrix A has specific forms. Consider as an example the Periodic Algorithm (Evans and Atkinson, 1970) which may be used when A has the form,

$$A = \begin{bmatrix} b_1 & c_1 & & & a_1 \\ a_2 & b_2 & c_2 & 0 & \\ & \ddots & \ddots & \ddots & c_{n-1} \\ & 0 & & a_n & b_n \\ c_n & & & & \end{bmatrix} \qquad (2.4.25)$$

The periodic algorithm, which is essentially Gauss Elimination, may be described as follows:

a) the elimination procedure

$$\left.\begin{aligned} w_1 &= 1/b_1 \\ g_1 = c_1 w_1, \quad h_1 &= a_1 w_1, \quad f_1 = d_1 w_1 \ , \\ G_1 = c_n, \quad D_1 &= b_n \ , \text{ and } F_1 = d_n \ , \end{aligned}\right\} \qquad (2.4.26)$$

then for i=2(1)n-1,

$$\left.\begin{aligned} w_i &= 1/(b_i - a_i g_{i-1}) \ , \\ g_i = c_i w_i, \quad h_i &= -h_{i-1} a_i w_i, \quad f_i = (d_i - a_i f_{i-1}) w_i \\ G_i = -g_{i-1} G_{i-1}, \quad D_i &= D_{i-1} - G_{i-1} h_{i-1} \quad \text{and} \quad F_i = F_{i-1} - G_{i-1} f_{i-1} \end{aligned}\right\}, \qquad (2.4.27)$$

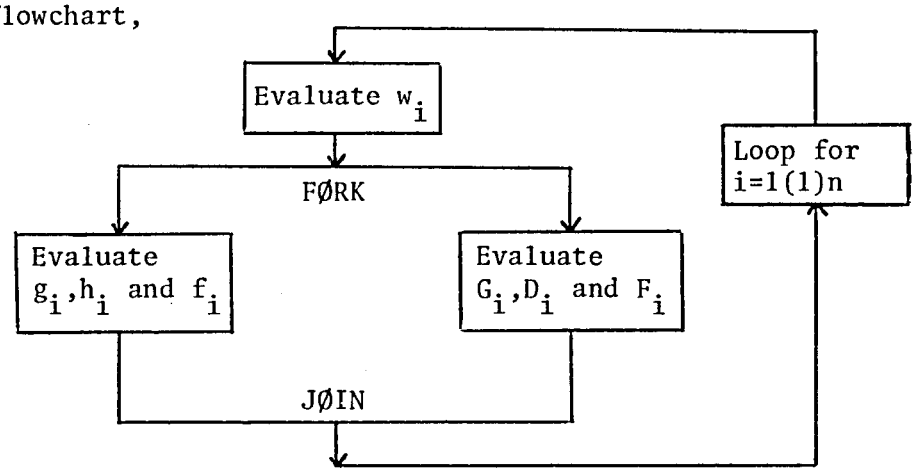
and finally,

$$\left.\begin{aligned} g_n &= h_n = G_n = F_n = 0 \\ D_n &= D_{n-1}-(G_{n-1}+a_n)(g_{n-1}+h_{n-1}) \quad \text{and} \quad f_n = F_{n-1}-(G_{n-1}+a_n)f_{n-1} \end{aligned}\right\} \tag{2.4.28}$$

b) the backward substitution process,

$$\left.\begin{aligned} x_n &= f_n/D_n \\ \text{and for } i=n-1(-1)1, & \\ x_i &= f_i-g_i x_{i+1}-h_i x_n \quad . \end{aligned}\right\} \tag{2.4.29}$$

The evaluation of the six quantities g_i, h_i, f_i, G_i, D_i and F_i are independent for each value of i and so may be computed simultaneously. Clearly a maximum of 6 processors may be used. One method of executing the factorisation procedure using 2 processors is given in the following flowchart,



and the speed-up and efficiency achieved are,

$$\left.\begin{aligned} S_2 &= \frac{(n-1)D+(9n-13)M+(4n-3)A}{(n-1)D+2(3n-4)M+(2n-1)A} < 2 \\ \text{and} \quad E_2 &= \frac{S_2}{2} < 1 \end{aligned}\right\} \tag{2.4.30}$$

The substitution process exhibits little inherent parallelism except for the calculation of the products $g_i x_{i+1}$ and $h_i x_n$. The parallel overhead incurred by forming these products simultaneously however would greatly reduce the speed-up that might be achieved and so the process should be executed sequentially.

This concludes the survey of numerical algorithms for inherent parallelism. The speed-ups that may be achieved by the exploitation of this parallelism appear to be very impressive. It must be realised however that the overheads incurred by the fork and join statements have not been taken into account. Although the effect of 1 fork and join is insignificant, the triangularisation procedure of Gaussian Elimination, for instance, has (n-1) steps, each requiring 2 sets of fork and join statements. The parallel overheads therefore will have a considerable effect on the speed-up achieved by these algorithms. In the algorithms that appear in the following chapters an attempt is made to minimise the parallel overheads by using as few fork and join statements as possible.

On a data flow computer, the machine code output by the compiler is a data flow language in which all results from instructions are linked to all successor instructions that require that result. Thus, an instruction is 'ready for execution' when all the data that it requires is available and parallelism is achieved by more than one instruction being ready for execution. This approach to parallelism reduces the inter-dependancy of processors because only instructions with the required data are executed.

Clearly, parallelism is introduced into data flow programs by the compiler and so existing software can be re-compiled into data flow form. However, rewriting this software in a high level data flow language can enhance the parallelism (Rumbaugh, 1977). The parallel algorithms developed in this thesis are potentially applicable for use on data flow parallel processors.

CHAPTER 3

THE PARALLEL SOLUTION OF BANDED SYSTEMS OF LINEAR EQUATIONS BY TRIANGULAR FACTORISATION

3.1 INTRODUCTION

A frequently occurring problem in the numerical solution of partial and ordinary differential equations is that of solving the banded system of equations

$$A\underline{x} = \underline{d} \quad , \tag{3.1.1}$$

where A is an (n×n) matrix of semi-bandwidth m, i.e. see (3.2.14). The importance of this problem in engineering applications emphasises the need to be able to solve it efficiently on a parallel computer.

Standard methods for the solution of linear systems such as Gaussian Elimination and Triangular Factorisation are presented in Chapter 2 and the derivation of parallel algorithms by the algorithm decomposition principle [Hyafil and Kung, 1974] are also outlined. Although the theoretical results for these methods are encouraging, their implementation on a parallel computer would not be so successful, since the time overhead incurred by the large number of 'forks' and 'joins' that are necessary in the program would degrade the performance of the algorithms. It is clear that a new strategy is required in order that we may solve the system (3.1.1) in parallel.

The folding algorithm of Evans and Hatzopoulos [1976] is based on the technique of performing Gaussian elimination in the top left and bottom right hand corners of A concurrently. In the following analysis a similar strategy is applied to Triangular Factorisation. Instead of upper and lower triangular matrices, the factorisation produces two matrices that are upper triangular in one half and lower triangular in the other half and vice versa.

Initially we consider the case where the matrix A is tridiagonal and present algorithms for unsymmetric and symmetric matrices. These algorithms are then expanded to solve the more general banded system (3.1.1). In these generalised algorithms, the matrices produced by

the factorisation process are seen to have an area of overlap at their centres that correspond to the interference that occurs in the folding algorithm.

3.2 STANDARD FACTORISATION ALGORITHMS

In this section standard factorisation methods are outlined so that they may be compared with the new parallel factorisation methods. First we consider the case of the tridiagonal system of equations.

Let the matrix A be an (n×n) matrix of the form:

$$\begin{bmatrix} a_1 & c_2 & & & & \\ b_2 & a_2 & c_3 & & 0 & \\ & b_3 & a_3 & c_4 & \ddots & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & b_{n-1} & a_{n-1} & c_n \\ & & & & b_n & a_n \end{bmatrix} \qquad (3.2.1)$$

with $\underline{x}$ and $\underline{d}$ as (n×1) vectors of the form:

$$\underline{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} , \qquad \underline{d} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} \qquad (3.2.2)$$

The triangular factorisation algorithm for system (3.2.1) involves determining triangular factors L and U such that,

$$A = L.U \qquad , \qquad (3.2.3)$$

where,

$$L = \begin{bmatrix} 1 & & & & \\ \ell_{21} & 1 & & & \\ & \ell_{31} & 1 & & \\ & & \ddots & \ddots & \\ & & & \ell_{n,n-1} & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} u_{11} & u_{12} & & & \\ & u_{22} & u_{23} & & \\ & & \ddots & \ddots & \\ & & & \ddots & u_{n-1,n} \\ & & & & u_{n,n} \end{bmatrix} \qquad (3.2.4)$$

This can be shown to be achieved by the following formulae:

$$\left.\begin{aligned} u_{11} &= a_1 \\ \ell_{i,i-1} &= \frac{b_i}{u_{i-1,i-1}} \\ u_{ii} &= a_i - \ell_{i,i-1}\, u_{i-1,i} \\ q_{i-1,i} &= c_i \end{aligned}\right\} \text{ for } i=2(1)n \qquad (3.2.5)$$

The system

$$L\underline{y} = \underline{d} \quad , \qquad (3.2.6)$$

is solved by a forward substitution process as follows,

$$\left.\begin{aligned} y_1 &= d_1 \quad , \\ y_i &= d_i - \ell_{i,i-1} \cdot y_{i-1} \quad , \text{ for } i=2(1)n \end{aligned}\right\} \qquad (3.2.7)$$

and the system,

$$U\underline{x} = \underline{y} \quad , \qquad (3.2.8)$$

is solved by a backward substitution process defined by,

$$\left.\begin{aligned} x_n &= \frac{y_n}{u_{nn}} \\ x_i &= (y_i - u_{i,i+1} x_{i+1})/u_{ii} \quad , \text{ for } i=n-1(-1)1 \ . \end{aligned}\right\} \qquad (3.2.9)$$

Adopting the standard technique of overwriting L and U on A to save computer storage, it is clear that the factorisation process (3.2.5) requires 2(n-1) multiplications and (n-1) additions and the forward and backward substitution processes (3.2.7) and (3.2.9) require (n-1) multiplications and (n-1) additions and (2n-1) multiplications and (n-1) additions respectively. Thus, the complete algorithm requires (5n-4) multiplications and 3(n-1) additions giving a total of (8n-7) arithmetic operations.

Obviously for the factorisation (3.2.3) to exist we require that the matrix A is positive definite, i.e., all of its eigenvalues are positive. Should this not be true then it is necessary to introduce partial pivoting as proposed in Wilkinson [1965].

The factorisation of (3.2.3), incorporating partial pivoting is now defined as follows.

At the i^{th} step of the evaluation of L and U form the quantities:

$$R_t = a_{t,i} - (\ell_{t,i-1}u_{i-1,t} + \ell_{t,i-2}u_{i-2,t}) \qquad (3.2.10)$$

for t=i,i+1 .

When $|R_{i+1}| > |R_i|$ (where $|z|=z$, if $z \geq 0$ otherwise $-z$), the rows i and i+1 are interchanged including the values of R_t and d_t.

Then, we have

$$\left.\begin{aligned} u_{i,i} &= R_i \quad , \\ \ell_{i+1,i} &= R_{i+1}/u_{i,i} \quad , \\ u_{i,i+1} &= a_{i,i+1} - \ell_{i,i-1}u_{i-1,i+1} \\ \text{and} \quad u_{i,i+2} &= a_{i,i+2} \quad . \end{aligned}\right\} \qquad (3.2.11)$$

The two substitution stages are now defined thus,

$$\left.\begin{aligned} y_1 &= d_1 \quad , \\ y_i &= d_i - \sum_{k=1}^{i-1} \ell_{i,k}y_k \quad , \text{ for } i=2(1)n \ , \end{aligned}\right\} \qquad (3.2.12)$$

and

$$\left.\begin{aligned} x_n &= y_n/u_{n,n} \\ x_{n-1} &= (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1} \\ x_i &= (y_i - (u_{i,i+1}x_{i+1} + u_{i,i+2}x_{i+2}))/u_{i,i}, \text{ for } i=n-2(-1)1. \end{aligned}\right\} \qquad (3.2.13)$$

Excluding comparisons and interchanges the algorithm requires $(n^2+17n-28)/2$ multiplications and $(n^2+13n-26)/2$ additions giving a total of $(n^2+15n-27)$ arithmetic operations.

If we now consider the general banded system (3.1.1) where A has the form:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} & & & \\ a_{21} & a_{22} & \ddots & \ddots & \ddots & & \\ \vdots & & & & & a_{n-m+1,n} & \\ a_{m,1} & \ddots & \ddots & \ddots & \ddots & \vdots & \\ & \ddots & \ddots & \ddots & \ddots & a_{n,n} & \\ & & a_{n,n-m+1} & \cdots & & & \end{bmatrix} \qquad (3.2.14)$$

then the factorisation of equation (3.2.3) where L is lower triangular and U upper triangular is as follows:

at the i^{th} step in the evaluation of L and U, form the quantities:

$$R_t = a_{t,i} - \sum_{j=\alpha}^{i-1} \ell_{t,j} u_{j,i}, \quad \text{for } t=i(1)i+(m-1) . \tag{3.2.15}$$

Select the maximum $|R_t|$ (let it be t'), and when $t' \neq i$, interchange rows t' and i of A. Then,

$$\left.\begin{aligned} u_{ii} &= R_i \quad , \\ \ell_{i+k,i} &= R_{i+k}/u_{i,i}, \quad \text{for } k=1(1)m-1. \\ \text{and} \quad q_{i,i+k} &= a_{i,i+k} - \sum_{j=\alpha}^{i-1} \ell_{i,j} u_{j,i+k}, \quad k=1(1)2(m-1) \\ \text{where} \quad \alpha &= \begin{cases} 1 & \text{for } i \leq 2m-k-1 \\ i-2(m-1)+k & \text{for } i>2m-k-1. \end{cases} \end{aligned}\right\} \tag{3.2.16}$$

The forward substitution process for the solution of $L\underline{y}=\underline{d}$ is as defined in (3.2.12) and the backward substitution process for the solution of $U\underline{x} = \underline{y}$ is now

$$\left.\begin{aligned} x_n &= y_n/u_{n,n} \\ x_i &= (y_i - (\sum_{k=i+1}^{\beta} u_{i,k} x_k))/u_{i,i}, \quad i=n-1(1)1 \\ \text{where} \quad \beta &= \begin{cases} n, & \text{for } i \geq n-2m+2 \\ i+2(m-1), & \text{for } i<n-2m+2. \end{cases} \end{aligned}\right\} \tag{3.2.17}$$

The execution of this algorithm requires $(3n^2+3n(8m^2-8m+1)-(m-1)(34m^2-29m+6))/6$ multiplications and $(3n^2+3n(8m^2-10m+1)-2(m-1)(17m^2-16m+3))/6$ additions, giving a total of $(n^2+n(m-1)(8m-1)-(m-1)(68m^2-61m+12)/6)$ arithmetic operations.

When the matrix A is symmetric and positive definite, it is possible to use the Choleski factorisation method where matrix A is factorised such that,

$$A = L.L^T \quad . \tag{3.2.18}$$

An advantage of this method is that it is only necessary to evaluate and store matrix L.

For the case when A is tridiagonal as in (3.2.1) we have that

$$b_i = c_i \qquad \text{for } i=2(1)n \tag{3.2.19}$$

and

$$L = \begin{bmatrix} \ell_{11} & & & & \\ \ell_{21} & \ell_{22} & & & \\ & \ell_{32} & \ell_{33} & & 0 \\ & & \ddots & \ddots & \\ & 0 & & \ell_{n,n-1} & \ell_{n,n} \end{bmatrix}, \tag{3.2.20}$$

where the $\ell_{i,j}$'s are defined as follows:

$$\left.\begin{aligned} \ell_{11} &= \sqrt{a_1} \\ \ell_{i,i-1} &= b_i/\ell_{i-1,i-1} \\ \text{and} \quad \ell_{i,i} &= \sqrt{a_i-\ell^2_{i,i-1}} \end{aligned}\right\} \quad \text{for } i=2(1)n \tag{3.2.21}$$

Then, the two substitution stages are defined to be,

$$\left.\begin{aligned} L\underline{y} &= \underline{d} \quad , \\ y_1 &= d_1/\ell_{11} \\ y_i &= (d_i-\ell_{i,i-1}y_{i-1})/\ell_{i,i} \quad \text{for } i=2(1)n \end{aligned}\right\} \tag{3.2.22}$$

and

$$\left.\begin{aligned} L^T\underline{x} &= \underline{y} \\ x_n &= y_n/\ell_{n,n} \\ x_i &= (y_i-\ell_{i+1,i}x_{i+1})/\ell_{i,i} \quad \text{for } i=n-1(1)1. \end{aligned}\right\} \tag{3.2.23}$$

The total number of arithmetic operations required by this algorithm is (10n-7), made up of n square roots, 2(3n-2) multiplications and 3(n-1) additions.

Finally we apply Choleski factorisation to (3.1.1) where A has the form (3.2.14) and $a_{i,j}=a_{j,i}$. The factorisation can be defined now as follows:

$$\left.\begin{aligned} \ell_{ii} &= \sqrt{(a_{ii} - \sum_{j=1}^{\alpha} \ell_{i,i-j}^2)} \\ \text{and} \quad \ell_{i+k,i} &= (a_{i+k,i} - \sum_{j=1}^{\alpha} \ell_{i+k,i-j}\ell_{i,i-j})/\ell_{ii}, \text{ for } k=1(1)m-1 \end{aligned}\right\} \text{for } i=1(1)n$$

$$\text{where} \quad \alpha = \begin{cases} i-1, & \text{for } i<(m-k) \\ m-k-1 & \text{for } i\geq(m-k) \end{cases} \qquad (3.2.24)$$

and the two substitution stages become,

$$\left.\begin{aligned} L\underline{y} &= \underline{d} \\ y_1 &= d_1/\ell_{11} \\ y_i &= (d_i - \sum_{k=1}^{\alpha} \ell_{i,i-k}y_{i-k})/\ell_{ii} \quad \text{for } i=2(1)n, \end{aligned}\right\} \qquad (3.2.25)$$

$$\text{where} \quad \alpha = \begin{cases} i-1 & \text{for } i<m \\ m-1 & \text{for } m\geq i \end{cases}$$

and

$$\left.\begin{aligned} L^T\underline{x} &= \underline{y} \\ x_n &= y_n/\ell_{nn} \\ x_i &= (y_i - \sum_{k=1}^{\beta} \ell_{i,i+k}x_{i+k})/\ell_{ii}, \text{ for } i=n-1(-1)1 \end{aligned}\right\} \qquad (3.2.26)$$

$$\text{where} \quad \beta = \begin{cases} n-i & \text{for } n-i+1<m \\ m-1 & \text{for } n-i+1\geq m \end{cases}$$

The total number of arithmetic operations required by this algorithm is $n(m^2+4m-2)-m(m-1)(4m+13)/6$, made up of n square roots, $n(m^2+5m-2)/2-m(m-1)(m+4)/3$ multiplications and $n(m-1)(m+4)/2-m(m-1)(2m+5)/6$ additions.

3.3 THE PARALLEL TRIANGULAR FACTORISATION OF THE MATRIX A

Let us now consider the tridiagonal system of linear equations (3.1.1) where A has the form (3.2.1). By applying the technique of

folding, we may factorise A into two matrices P and Q such that

$$A = P.Q \quad , \tag{3.3.1}$$

where P has the form:-

$$\begin{bmatrix} 1 & & & & & & \\ p_{2,1} & 1 & & & & 0 & \\ & p_{3,2} & 1 & & & & \\ & & \ddots & \ddots & & & \\ & & & p_{s,s-1} & 1 & p_{s,s+1} & \\ & & 0 & & \ddots & \ddots & \\ & & & & & 1 & p_{n-1,n} \\ & & & & & & 1 \end{bmatrix} \tag{3.3.2}$$

and Q the form:

$$\begin{bmatrix} q_{1,1} & q_{1,2} & & & & & \\ & q_{2,2} & q_{2,3} & & & 0 & \\ & & \ddots & \ddots & & & \\ & & & q_{s-1,s-1} & q_{s-1,s} & & \\ & & & & q_{s,s} & & \\ & & 0 & & q_{s+1,s} & q_{s+1,s+1} & \\ & & & & & \ddots & \ddots \\ & & & & & & q_{n,n-1} \quad q_{n,n} \end{bmatrix} \tag{3.3.3}$$

where

$$s = \begin{cases} \frac{n+1}{2} & , \text{ for n is odd} \\ \frac{n}{2} & , \text{ for n is even} \end{cases} \tag{3.3.4}$$

Since by definition we have,

$$A = P.Q \quad ,$$

then on substitution into (3.1.1) we have the following system to solve,

$$P.Q\underline{x} = \underline{d} \quad . \tag{3.3.5}$$

In order to solve the given tridiagonal system (3.1.1) we introduce an auxiliary vectory $\underline{y}$ such that

$$Q\underline{x} = \underline{y} \quad .$$

Therefore, the problem reduces to that of solving the two systems

$$P\underline{y} = \underline{d} \quad \text{and} \quad Q\underline{x} = \underline{y} \,. \tag{3.3.6}$$

To evaluate the elements $p_{i,j}$ and $q_{r,s}$ of the matrices P and Q we form the product P.Q which is given by,

$$P.Q \equiv \begin{bmatrix} q_{11} & q_{12} & & & & \\ p_{21}q_{11} & (p_{21}q_{12}+q_{22}) & q_{23} & & & 0 \\ & [& \gamma &] & & \\ & & & & & \\ 0 & & q_{n-1,n-2} & (q_{n-1,n-1}+p_{n-1,n}q_{n,n-1}) & p_{n-1,n}q_{n,n} \\ & & & q_{n,n-1} & q_{n,n} \end{bmatrix} \tag{3.3.7}$$

where the submatrix γ is defined as

$$\gamma = [(p_{s,s-1}q_{s-1,s-1})\,(p_{s,s-1}q_{s-1,s}+q_{s,s}+p_{s,s+1}q_{s+1,s})\,(p_{s,s+1}q_{s+1,s+1})]. \tag{3.3.8}$$

On equating the matrices A and P.Q, we derive the following relationships:

$$\left.\begin{array}{rlrl} q_{11} &= a_1 & \text{and} \quad q_{n,n} &= a_n \\ q_{12} &= c_2 & q_{n,n-1} &= b_n \\ p_{21}q_{11} &= b_2 & p_{n-1,n}q_{n,n} &= c_n \\ p_{21}q_{12}+q_{22} &= a_2 & q_{n-1,n-1}+p_{n-1,n}q_{n,n-1} &= a_{n-1} \\ \cdots\cdots & & \cdots\cdots & \end{array}\right\} \tag{3.3.9}$$

Using these equations we can obtain the following formulae to establish the unknown quantities $p_{i,j}$ and $q_{r,s}$ respectively:

$$q_{11} = a_{11} \qquad\qquad q_{n,n} = a_n$$

$$p_{i+1,i} = \frac{b_{i+1}}{q_{i,i}}, \quad \text{for } i=1(1)s-1 \qquad p_{i-1,i} = \frac{c_i}{q_{i,i}}, \quad \text{for } i=n(-1)s+1$$

$$q_{i,i+1} = c_{i+1}, \quad \text{for } i=1(1)s-1 \qquad q_{i,i-1} = b_i, \quad \text{for } i=n(-1)s+1$$

$$q_{i,i} = a_i - p_{i,i-1}q_{i-1,i}, \quad \text{for } i=2(1)s-1 \qquad q_{i,i} = a_i - p_{i,i+1}q_{i+1,i}, \quad \text{for } i=n-1(-1)s+1 \tag{3.3.10}$$

and finally,

$$q_{s,s} = a_s - (p_{s,s-1}q_{s-1,s} + p_{s,s+1}q_{s+1,s}) . \tag{3.3.11}$$

With the matrices P and Q known, the given tridiagonal system (3.1.1) is reduced to solving $P\underline{y}=\underline{d}$ for $\underline{y}$ using an inward substitution process i.e., a forward substitution from the top left hand corner of P and a backward substitution from the bottom right hand corner of P intersecting at its centre point, followed by solving $Q\underline{x}=\underline{y}$ for $\underline{x}$ using an outward substitution process i.e., a backward substitution from the centre point to the top left hand corner of Q and a forward substitution from the centre point to the bottom right hand corner of the matrix Q.

3.4 PARALLEL TRIANGULAR FACTORISATION WITH PARTIAL PIVOTING

As in the standard factorisation methods, for the factorisation of (3.3.1) to exist, matrix A must be positive definite. If this condition is not satisfied, we have to introduce the equivalent strategy of partial pivoting as proposed by Wilkinson [1965]. The new factorisation procedure is then defined as follows:-

at the i^{th} step in the evaluation of P and Q where i=1(1)s-1 form the quantities,

$$R_t = a_{t,i} - (p_{t,i-1}q_{i-1,i} + p_{t,i-2}q_{i-2,i}) \quad \text{for } t=i,i+1 . \tag{3.4.1}$$

If $|R_{i+1}|>|R_i|$, the rows i and i+1 are interchanged including R_t and d_t.

We then have

$$\left.\begin{aligned} q_{i,i} &= R_i \\ p_{i+1,i} &= \frac{R_{i+1}}{q_{i,i}} \\ q_{i,i+1} &= a_{i,i+1}-p_{i,i-1}q_{i-1,i+1} \\ \text{and} \quad q_{i,i+2} &= a_{i,i+2} \end{aligned}\right\} \qquad (3.4.2)$$

Similarly at the i^{th} step in the evaluation of P and Q, when i=n(-1)s+2 form the quantities,

$$R_t = a_{t,i}-(p_{t,i+1}q_{i+1,i}+p_{t,i+2}q_{i+2,i}) \text{ for } t=i,i-1. \qquad (3.4.3)$$

Again if $|R_{i-1}|>|R_i|$ then the rows i and i-1 are interchanged including R_t and d_t.

Then,

$$\left.\begin{aligned} q_{i,i} &= R_i \\ p_{i-1,i} &= \frac{R_{i-1}}{q_{i,i}} \\ q_{i,i-1} &= a_{i,i-1}-p_{i,i+1}q_{i+1,i-1} \\ \text{and} \quad q_{i,i-2} &= a_{i,i-2} \end{aligned}\right\} \qquad (3.4.4)$$

Finally, at the centre, we have

$$\left.\begin{aligned} R_{s+1} &= a_{s+1,s+1}-(p_{s+1,s+2}q_{s+2,s+1}+p_{s+1,s+3}q_{s+3,s+1}+ \\ &\qquad p_{s+1,s-1}q_{s-1,s+1}) \\ R_s &= a_{s,s+1}-(p_{s,s+2}q_{s+2,s+1}+p_{s,s+3}q_{s+3,s+1}+ \\ &\qquad p_{s,s-1}q_{s-1,s+1}) \end{aligned}\right\} \qquad (3.4.5)$$

If $|R_s|>|R_{s+1}|$ then interchange the rows s and s+1 and we then have:

$$\left.\begin{aligned} q_{s+1,s+1} &= R_{s+1} \\ p_{s,s+1} &= \frac{R_s\ R_s}{q_{s+1,s+1}} \\ q_{s+1,s} &= a_{s+1,s}-(p_{s+1,s-2}q_{s-2,s}+p_{s+1,s-1}q_{s-1,s}+p_{s+1,s+2}q_{s+2,s}) \\ \text{and} \quad q_{s,s} &= a_{s,s}-(p_{s,s-1}q_{s-1,s}+p_{s,s-2}q_{s-2,s}+p_{s,s+1}q_{s+1,s}+p_{s,s+2}q_{s+2,s}) \end{aligned}\right\} \qquad (3.4.6)$$

Note that in the top half of the system, any row can be interchanged upwards only once yet any row can be interchanged downwards as far as row s+1. Similarly in the lower half of the system a row may be interchanged downwards only once but upwards as far as row s.

Matrices P and Q will now have the form:

$$P = \begin{bmatrix} 1 & & & & & & \\ p_{21} & 1 & & & 0 & & \\ & \ddots & \ddots & & & & \\ & & & p_{s,s-1} & 1 & p_{s,s+1} & \cdots & p_{s,n} \\ p_{s+1,1} & \cdots & p_{s+1,s-1} & & & 1 & & \vdots \\ & & & & & & \ddots & p_{n-1,n} \\ & & 0 & & & & & 1 \end{bmatrix} \qquad (3.4.7)$$

and

$$Q = \begin{bmatrix} q_{11} & q_{12} & q_{13} & & & & & & \\ & q_{22} & q_{23} & q_{24} & & & & 0 & \\ & & \ddots & \ddots & \ddots & & & & \\ & & & q_{s-1,s-1} & q_{s-1,s} & q_{s-1,s+1} & & & \\ & & & & q_{s,s} & & & & \\ & & & & q_{s+1,s} & q_{s+1,s+1} & & & \\ & & 0 & & q_{s+2,s} & & \ddots & & \\ & & & & & & \ddots & \ddots & \\ & & & & & & & q_{n,n-2} & q_{n,n-1} & q_{n,n} \end{bmatrix} \qquad (3.4.8)$$

A comment on these new matrices P and Q is that, with regard to P, there are only (n-1) elements apart from the diagonal elements such that $p_{i,j} \neq 0$. However, because the pivoting process includes the interchanging of the $p_{i,j}$'s, the non-zero $p_{i,j}$'s will be dispersed over the area indicated in (3.4.7). With regard to Q, there is a maximum of 3(n-1) elements such that $q_{r,s} \neq 0$ and it is possible that of the non-zero elements indicated in (3.4.8) a proportion of the off-diagonal elements may be zeros.

3.5 THE SOLUTION OF THE SYSTEM (3.1.1) BY THE PARALLEL TRIANGULAR FACTORISATION METHOD

The method is characterised by the inward and outward substitution processes which we describe as follows:-

a) The inward substitution is given by the matrix system,

$$P\underline{y} = \underline{d} \quad .$$

In particular we have two processes; a forward substitution process starting from the top left hand corner, i.e.,

$$y_1 = d_1 \tag{3.5.1}$$

and

$$y_i = \begin{cases} d_i - \sum_{k=1}^{i-1} p_{i,k}y_k & \text{with pivoting} \\ d_i - p_{i,i-1}y_{i-1} & \text{without pivoting,} \end{cases} \quad \text{for } i=2(1)s-1 \tag{3.5.2}$$

with a backward substitution process from the bottom right hand corner, i.e.,

$$y_n = d_n \tag{3.5.3}$$

$$y_i = \begin{cases} d_i - \sum_{k=i+1}^{n} p_{i,k}y_k \text{ , for } i=n-1(-1)s+2 & \text{with pivoting} \\ d_i - p_{i,i+1}y_{i+1} \text{ , for } i=n-1(-1)s+1 & \text{without pivoting} \end{cases} \tag{3.5.4}$$

and

$$\left.\begin{aligned} y_s &= d_s-(p_{s,s-1}y_{s-1}+p_{s,s+1}y_{s+1}), && \text{without pivoting} \\ y_{s+1} &= d_{s+1}-(\sum_{k=1}^{s-1} p_{s+1,k}y_k + \sum_{k=s+2}^{n} p_{s+1,k}y_k) \\ y_s &= d_s-(\sum_{k=1}^{s-1} p_{s,k}y_k + \sum_{k=s+1}^{n} p_{s,k}y_k) \end{aligned}\right\} \text{ with pivoting} \tag{3.5.5}$$

b) The outward substitution is given by the matrix system:

$$Q\underline{x} = \underline{y} \ ,$$

or in point form,

$$x_s = \frac{y_s}{q_{s,s}} \tag{3.5.6}$$

$$x_{s+1} = (y_{s+1}-q_{s+1,s}x_s)/q_{s+1,s+1} \quad \text{with pivoting only} \tag{3.5.7}$$

and the backward substitution from the centre to the top left hand corner is given by:

$$x_i = \begin{cases} (y_i - q_{i,i+1}x_{i+1})/q_{i,i} , & \text{without pivoting} \\ (y_i - (q_{i,i+1}x_{i+1} + q_{i,i+2}x_{i+2}))/q_{i,i}, & \text{with pivoting} \end{cases} \quad \text{for } i=s-1(-1)1, \tag{3.5.8}$$

while the forward substitution from the centre to the bottom right hand corner is given by:

$$x_i = \begin{cases} (y_i - q_{i,i-1}x_{i-1})/q_{i,i} , & \text{for } i=s+1(1)n \quad \text{without pivoting} \\ (y_i - (q_{i,i-1}x_{i-1} + q_{i,i-2}x_{i-2}))/q_{i,i}, & \text{for } i=s+2(1)n \quad \text{with pivoting} \end{cases} \tag{3.5.9}$$

3.6 THE INHERENT PARALLELISM OF THE METHOD

The Parallel Triangular Factorisation Method, like the standard factorisation methods of section (3.2), comprises of three stages, i.e., the factorisation of A, the solution of $P\underline{y}=\underline{d}$ by an inward substitution process and the solution of $Q\underline{x}=\underline{y}$ by an outward substitution process. Examining each stage in turn we have:

1) The factorisation of A.

Clearly, the factorisation processes of (3.3.10) and (3.4.1) to (3.4.4) can be divided into two phases that are independent of each other, and so they may be executed concurrently. When these phases have been completed, the evaluation of the central elements [(3.3.11) and (3.4.5)-(3.4.6)] may be done.

The following diagram shows the order of evaluation when pivoting is not included:

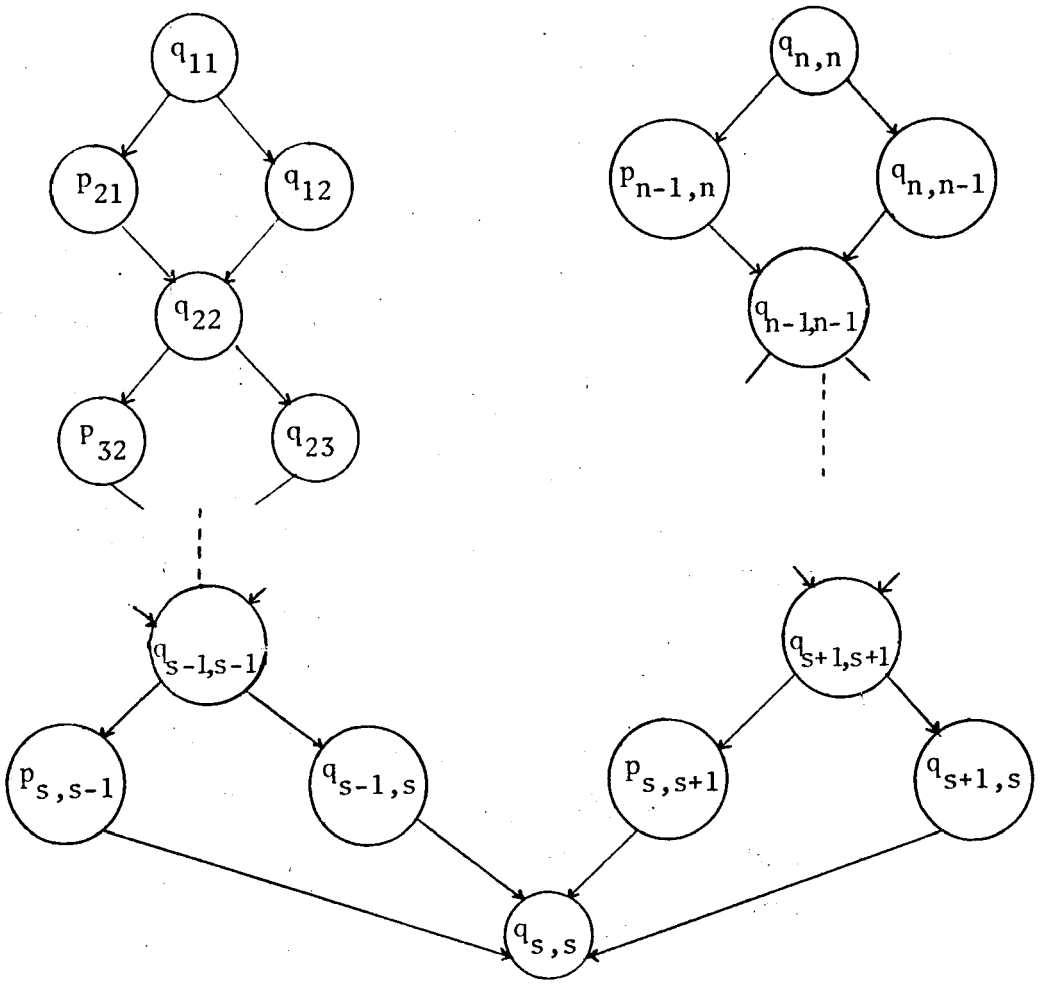


FIGURE 3.1

It is clear that up to 4 processors may be used concurrently.

2) Solution of $P\underline{y}=\underline{d}$.

Since the derivation of $\underline{y}$ is in fact a forward substitution process and a backward substitution process which are independent of each other, they may be executed in parallel.

Also, since the order of evaluation of the p's is the same as the order in which they are required for solving $P\underline{y}=\underline{d}$, the two processes may be done in parallel if the solution of $P\underline{y}=\underline{d}$ is set one step or evaluation out of phase.

c) Solution of $Q\underline{x}=\underline{y}$.

As before since the derivation of $\underline{x}$ involves two independent substitution processes, then they may be executed in parallel.

The flowchart for the parallel factorisation method without pivoting when implemented on a two processor system is given in Figure 3.2.

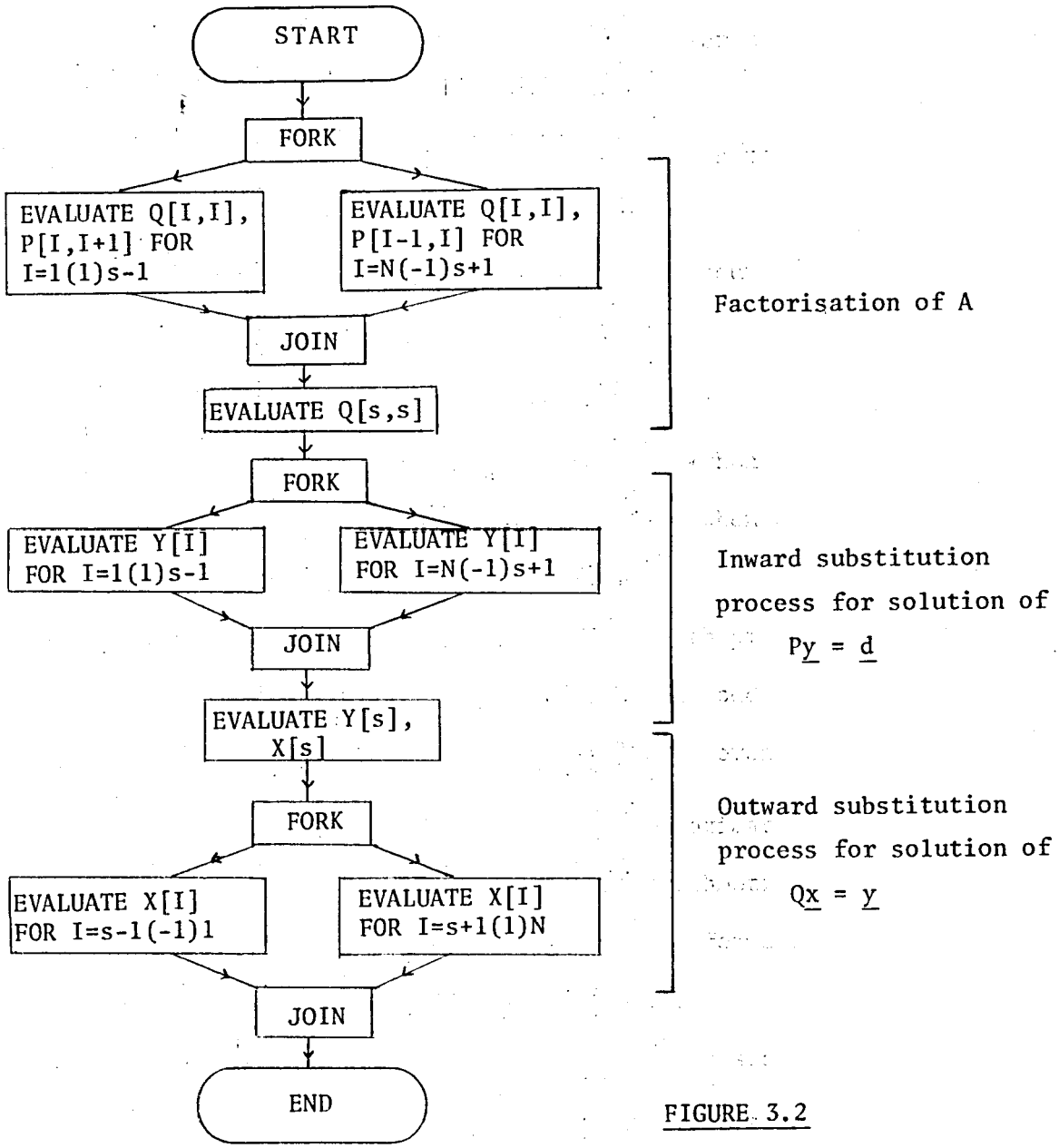


FIGURE 3.2

The phases of the algorithm that permit parallel processing are preceded by a 'FØRK' and followed by a 'JØIN' in Figure 3.2, and clearly, one processor is assigned to each branch of the phase.

From Figure 3.2 it is clear that the execution time for the whole algorithm is the sum of the execution times for the sequential and parallel phases. The sequential phases of the algorithm, which may be executed by either processor, can be treated as a sequential algorithm. For parallel phases however, each path is treated as a sequential path, and then the execution time for the phase becomes equal to that of the longest path.

Thus, by calculating the execution times in terms of multiplication and addition operations, we have the following results:

for parallel triangular factorisation without pivoting,

$$T_2 = \begin{cases} (5n+1)/2.M + (3n+1)/2.A & , n \text{ odd} \\ (5n/2+3).M + (3n/2+2).A & , n \text{ even} \end{cases}$$

giving

$$S_2 = \begin{cases} 2-(10.M+8.A)/((5n+1).M+(3n+1).A) & \text{when n is odd} \\ 2-(20.M+14.A)/((5n+6).M+(3n+4).A) & \text{when n is even.} \end{cases}$$

Clearly, as n increases, $S_2 \to 2$, and

for parallel triangular factorisation with pivoting, and $M = A$

$$T_2 = \begin{cases} n^2/4+11n+15/4, & \text{when n is odd} \\ n^2/4+21n/2-3, & \text{when n is even.} \end{cases}$$

These results suggest a speed up of approximately 4 since the sequential algorithm requires $(n^2+15n-27)$ arithmetic operations. This is due to the n^2 factor which arises from the forward and corresponding inward substitution phases of the algorithms.

It has already been noted that matrix P is sparse, as is matrix L, and so the operations in these substitution stages are largely redundant. This may be overcome by incorporating the substitution stage into the factorisation stages. The immediate transference of the p's, and (in the sequential algorithm) the ℓ's, to the right hand side removes this large number of redundant operations.

So we now have for the sequential algorithm:

$$T_1 = (18n-29) \quad \text{arithmetic operations,}$$

and for the parallel algorithm, in terms of arithmetic operations:

$$T_2 = \begin{cases} 9n+10 \ , & \text{when n is odd} \\ 9n+1 \ , & \text{when n is even,} \end{cases}$$

giving

$$S_2 = \begin{cases} (18n-29)/(9n+10) = 2-49/(9n+10) & \text{when n is odd} \\ (18n-29)/(9n+1) = 2-31/(9n+1) & \text{when n is even.} \end{cases}$$

Once again it is clear that as n increases the speed-up approaches the limit 2.

3.7 THE SYMMETRIC PARALLEL FACTORISATION METHOD

When matrix A (3.2.1) is symmetric (i.e. $c_i=b_i$, i=2(1)n), we can perform the factorisation so that

$$Q = P^T \ . \tag{3.7.1}$$

Consequently P will be identical with (3.3.2) except that the leading diagonal will be $p_{i,i}$ (i=1(1)n) instead of unity values. Then Q will be identical to (3.3.3) with $q_{i,j}=p_{j,i}$.

a) Evaluation of the elements of P.

The product P.Q (or PP^T) is now

$$\begin{bmatrix} p_{11}^2 & p_{11}p_{21} & & & \\ p_{21}p_{11} & (p_{21}^2+p_{22}^2) & p_{22}p_{32} & & 0 \\ & p_{32}p_{22} & \ddots & \ddots & \\ & & \ddots & (p_{n-1,n-1}^2+p_{n-1,n}^2) & p_{n-1,n}p_{n,n} \\ 0 & & & p_{n,n}p_{n-1,n} & p_{n,n}^2 \end{bmatrix} \tag{3.7.2}$$

and again on equating with the elements of A we have the formulae for P as follows:

$$p_{11} = \sqrt{a_1}\ , \qquad p_{n,n} = \sqrt{a_n}\ ,$$

$$p_{i+1,i} = \frac{b_{i+1}}{p_{i,i}}\ , \text{ for } i=1(1)s-1, \qquad p_{i-1,i} = \frac{b_i}{p_{i,i}}\ , \text{ for } i=n(-1)s+1,$$

$$p_{i,i} = \sqrt{a_i - p_{i,i-1}^2}, \text{ for } i=2(1)s-1, \quad p_{i,i} = \sqrt{a_i - p_{i,i+1}^2}, \text{ for } i=n-1(-1)s+1,$$

and

$$p_{s,s} = \sqrt{a_s - (p_{s,s-1}^2 + p_{s,s+1}^2)} \qquad (3.7.3)$$

The parallel evaluation of the p's may now be computed in the order illustrated in Figure 3.3.

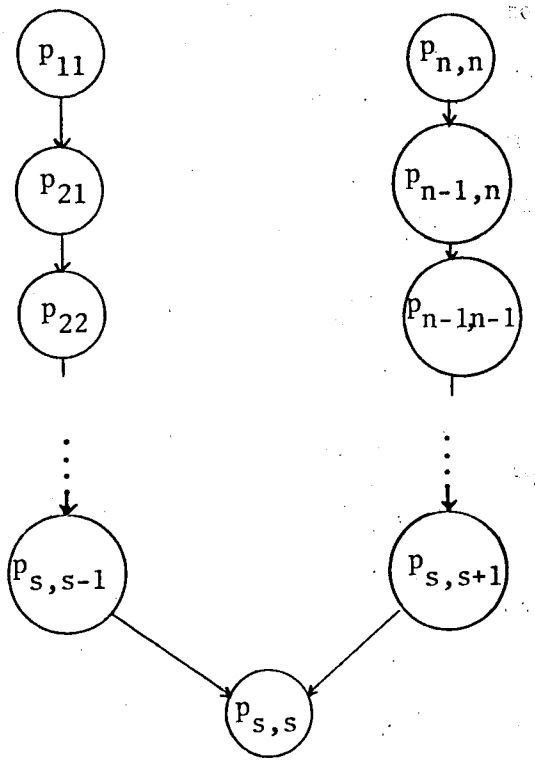


FIGURE 3.3

Partial pivoting may not, of course, be included in this method as the interchange of rows would upset the symmetry of the system.

b) The solution of $P\underline{y}=\underline{d}$.

The vector $\underline{y}$ is obtained as the solution of the system $P\underline{y}=\underline{d}$ in the following manner,

$$y_1 = \frac{d_1}{p_{11}}\ , \qquad y_n = \frac{d_n}{p_{n,n}}$$

$$y_i = (d_i - p_{i,i-1}y_{i-1})/p_{i,i}\ , \quad \text{for } i=2(1)s-1, \qquad y_i = (d_i - p_{i,i+1}y_{i+1})/p_{i,i}\ , \quad \text{for } i=n-1(-1)s+1, \tag{3.7.4}$$

and

$$y_s = (d_s - (p_{s,s-1}y_{s-1} + p_{s,s+1}y_{s+1}))/p_{s,s}\ .$$

c) The solution of $Q\underline{x}=\underline{y}$.

To obtain the final solution $\underline{x}$ we proceed as given by (3.5.6) to (3.5.9) without pivoting except that instead we use $q_{i,j}=p_{j,i}$.

When A is symmetric this method has the advantage of only having to evaluate and store matrix P in the computer memory.

If we consider the number of arithmetic operations required by the method where SR denotes square roots and M, A multiplication and addition times respectively, we have:

$$T_2 = \begin{cases} (n+1)/2 \cdot SR + (3n+1) \cdot M + (3n+1)/2 \cdot A & n \text{ odd} \\ (n/2+1) \cdot SR + (3n+4) \cdot M + (3n+4)/2 \cdot A & n \text{ even} \end{cases}$$

giving

$$S_2 = \begin{cases} 2 - (1 \cdot SR + 6 \cdot M + 4 \cdot A)/((n+1)/2 \cdot SR + (3n+1) \cdot M + (3n+1)/2 \cdot A) & n \text{ odd} \\ 2 - (2 \cdot SR + 12 \cdot M + 7 \cdot A)/((n/2+1) \cdot SR + (3n+4) \cdot M + (3n+4)/2 \cdot A) & n \text{ even.} \end{cases}$$

It is obvious that for large n, S_2 is again approximately 2.

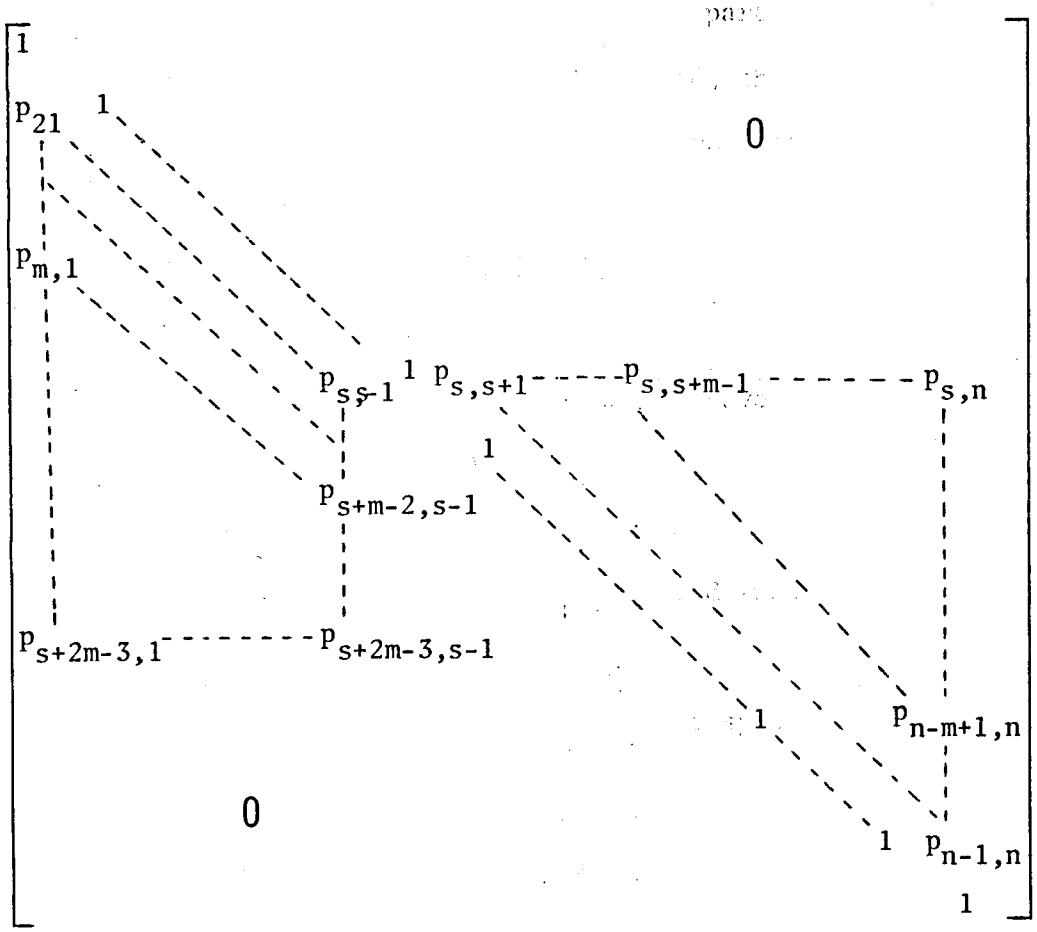
3.8 THE GENERALISATION OF THE METHODS FOR MATRICES OF SEMI-BANDWIDTH m

The two algorithms are now generalised to solve the same problem (3.1.1) but with matrix A having the form (3.2.14).

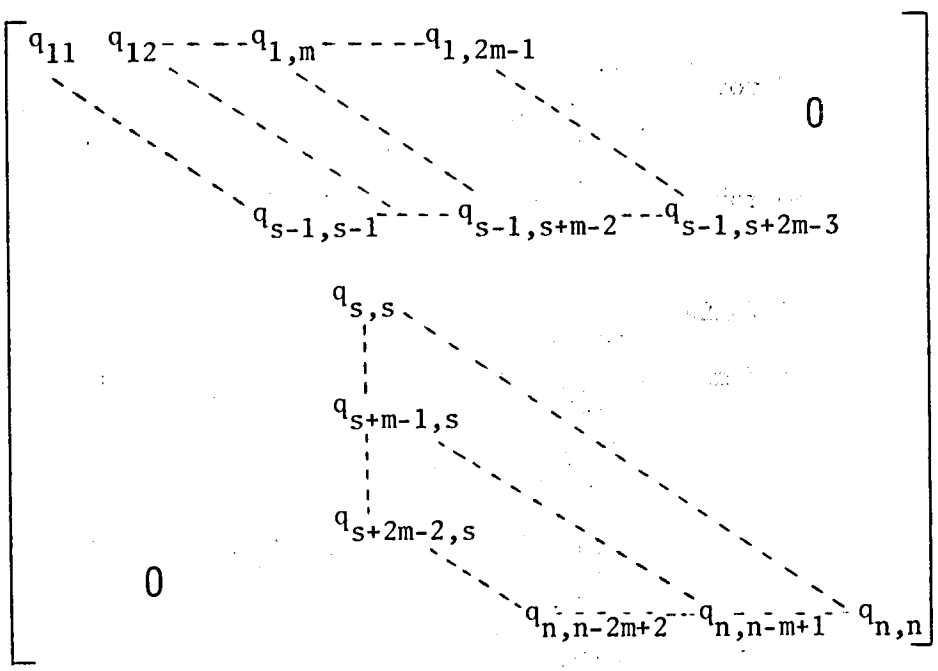
The main difficulty with the generalisation of these algorithms is that, when factorising matrix A, we are no longer left with two matrices that are purely upper triangular in one half and lower triangular in the other half, but two matrices which overlap at their centres corresponding to the interference area which occurs in the folding algorithm.

The Parallel Triangular Factorisation of A

Once again, matrix A is factorised into two matrices P and Q where P has the form:



and Q has the form: (3.8.1)



(3.8.2)

where $$s = \begin{cases} \frac{1}{2}n-m+2, & \text{for } n \text{ is even} \\ \frac{1}{2}(n+5)-m & \text{for } n \text{ is odd.} \end{cases} \quad . \qquad (3.8.3)$$

If, during the factorisation of matrix A, partial pivoting is not applied or if no rows of A are interchanged, the elements of P and Q will all be zero except for the shaded areas of (3.8.1) and (3.8.2).

As before, we have

$$A = P.Q \quad ,$$

(N.B. when partial pivoting is applied, A will have its rows permuted) which on substitution into (3.1.1) gives,

$$P.Q.\underline{x} = \underline{d} \quad .$$

Again, the auxiliary vector $\underline{y}$ is introduced such that

$$Q\underline{x} = \underline{y} \quad ,$$

thus reducing the problem to the solving of two systems of linear equations

$$P\underline{y} = \underline{d} \quad \text{and} \quad Q\underline{x} = \underline{y} \quad .$$

The matrix product P.Q is now defined as follows:-

<u>for $1 \leq i < s$</u>,

$$\left.\begin{aligned} (P.Q)_{i,i+k} &= q_{i,i+k} + \sum_{\ell=\alpha}^{i-1} p_{i,\ell} q_{\ell,i+k} \quad , \text{ for } 0 \leq k < 2(m-1) \\ (P.Q)_{i+k,i} &= p_{i+k,i} q_{i,i} + \sum_{\ell=1}^{i-1} p_{i+k,\ell} q_{\ell,i}, \text{ for } 0 < k < s+2m-i-3 \end{aligned}\right\} \qquad (3.8.4)$$

$$\alpha = \begin{cases} 1 & , \text{ for } i \leq 2m-k-1 \\ i-2(m-1)+k, & \text{for } i > 2m-k-1 \end{cases}$$

for $n \geq i \geq s+2(m-1)$

$$\left.\begin{aligned}
(P.Q)_{i,i-k} &= q_{i,i-k} + \sum_{\ell=i+1}^{\beta} p_{i,\ell} q_{\ell,i-k} && \text{for } 0 \leq k < 2(m-1) \\
(P.Q)_{i-k,i} &= p_{i-k,i} q_{i,i} + \sum_{\ell=i+1}^{n} p_{i-k,\ell} q_{\ell,i} && \text{for } 0 < k < i-s \\
\beta &= \begin{cases} n & , \text{ for } (n-i+1) \leq 2m-k-1 \\ i+2(m-1)-k, & \text{for } (n-i+1) > 2m-k-1, \end{cases}
\end{aligned}\right\} \quad (3.8.5)$$

for $s+2(m-1) > i \geq s$

$$\left.\begin{aligned}
(P.Q)_{i,i-k} &= q_{i,i-k} + \sum_{\ell=i+1}^{\beta} p_{i,\ell} q_{\ell,i-k} + \sum_{\ell=i-2(m-1)}^{s-1} p_{i,\ell} q_{\ell,i-k} \\
&\qquad \text{for } 0 \leq k \leq i-s \\
(P.Q)_{i-k,i} &= p_{i-k,i} q_{i,i} + \sum_{\ell=i+1}^{\beta} p_{i-k,\ell} q_{\ell,i} + \sum_{\ell=i-2(m-1)-k}^{s-1} p_{i-k,\ell} q_{\ell,i} \\
&\qquad \text{for } 0 < k \leq i-s
\end{aligned}\right\} \quad (3.8.6)$$

where β is as defined in (3.8.5).

The two matrices (P.Q) and A are equated so as to establish the unknown quantities $p_{i,j}$ and $q_{r,s}$. The full algorithm including partial pivoting is presented here.

So we have,

for $i=1(1)s-1$

we form the quantities,

$$\left.\begin{aligned}
R_t &= a_{t,i} - \sum_{\ell=\alpha}^{i-1} p_{t,\ell} q_{\ell,i} \quad , \quad \text{for } t = i(1)i+(m-1) \\
\alpha &= \begin{cases} 1 & , \text{ for } i \leq 2m-1 \\ i-2(m-1), & \text{for } i > 2m-1 \end{cases} .
\end{aligned}\right\} \quad (3.8.7)$$

Then select the maximum $|R_t|$ (let it be $|R_{t'}|$) and provided $t' \neq i$, we interchange rows t' and i including the values of R_t and d_t.

Then,

$$\left.\begin{aligned} q_{i,i} &= R_i \quad , \\ p_{i+k,i} &= \frac{R_{i+k}}{q_{i,i}} \quad , \quad \text{for } k=1(1)m-1 \\ \text{and} \quad q_{i,i+k} &= a_{i,i+k} - \sum_{\ell=\alpha}^{i-1} p_{i,\ell} q_{\ell,i+k} \quad , \quad \text{for } k=1(1)2(m-1) \end{aligned}\right\} \tag{3.8.8}$$

where α is defined in (3.8.4):

<u>for i=n(-1)s+2(m-1)</u>

we form the quantities,

$$\left.\begin{aligned} R_t &= a_{t,i} - \sum_{\ell=i+1}^{\beta} p_{t,\ell} q_{\ell,i}, \quad \text{for } t=i(-1)i-(m-1) \\ \beta &= \begin{cases} n & , \text{ for } (n-i+1) \leq 2m-1 \\ i+2(m-1), & \text{for } (n-i+1) > 2m-1 \ . \end{cases} \end{aligned}\right\} \tag{3.8.9}$$

Select the maximum $|R_t|$ (let it be $|R_{t'}|$) and again interchange rows t' and i including R_t and d_t provided $t' \neq i$.

Then, we form

$$\left.\begin{aligned} q_{i,i} &= R_i \ , \\ p_{i-k,i} &= \frac{R_{i-k}}{q_{i,i}} \quad , \quad \text{for } k=1(1)m-1 \\ \text{and} \quad q_{i,i-k} &= a_{i,i-k} - \sum_{\ell=i+1}^{\beta} p_{i,\ell} q_{\ell,i-k}, \quad \text{for } k=1(1)2(m-1) \end{aligned}\right\} \tag{3.8.10}$$

where β is defined in (3.8.5):

<u>for i=s+2m-3(-1)s</u>

we form the quantities,

$$R_t = a_{t,i} - \left(\sum_{\ell=i+1}^{\beta} p_{t,\ell} q_{\ell,i} + \sum_{\ell=i-2(m-1)}^{s-1} p_{t,\ell} q_{\ell,i} \right), \tag{3.8.11}$$

$$\text{for } t=i(-1)s$$

where β is defined in (3.8.9).

Again we select the maximum $|R_t|$, interchange the rows

accordingly and form,

$$\left.\begin{aligned} q_{i,i} &= R_i \quad , \\ p_{i-k,i} &= \frac{R_{i-k}}{q_{i,i}} \quad , \text{ for } k=1(1)i-s \\ \text{and finally,} \\ q_{i,i-k} &= a_{i,i-k} - \Big(\sum_{\ell=i+1}^{\beta} p_{i,\ell} q_{\ell,i-k} + \sum_{\ell=i-2(m-1)-k}^{s-1} p_{i,\ell} q_{\ell,i-k} \Big), \\ & \qquad \text{for } k=1(1)i-s \end{aligned}\right\} \tag{3.8.12}$$

where β is defined in (3.8.5).

In order to take full advantage of the accuracy of this method, double-precision accumulation of inner products such as $\sum pq$ should be used. If possible, the R_t values should only be rounded to single precision when the maximum $|R_t|$ has been selected.

With matrices P and Q known, once again $P\underline{y}=\underline{d}$ is solved for $\underline{y}$ using an inward substitution process and $Q\underline{x}=\underline{y}$ for $\underline{x}$ using an outward substitution process.

Note that now, in the top half of the system, a row may be interchanged upwards, only once, a maximum of (m-1) rows yet a row may be interchanged downwards as far as row (s+2m-3). Whilst in the lower half, a row may be interchanged downwards a maximum of (m-1) rows and upwards as far as row s.

3.9 THE SOLUTION OF THE GENERALISED SYSTEM BY THE PARALLEL TRIANGULAR FACTORISATION METHOD

The inward and outward substitution processes are now defined as follows:-

a) the inward substitution for the solution of

$$P\underline{y} = \underline{d} \ ,$$

where the forward substitution from the top left hand corner is given by,

$$\left.\begin{aligned} y_1 &= d_1 \\ y_i &= d_i - \sum_{\ell=1}^{i-1} p_{i,\ell} y_\ell \ , \end{aligned}\right\} \qquad (3.9.1)$$

for i=2(1)s-1

and the backward substitution from the bottom right hand corner is

$$\left.\begin{aligned} y_n &= d_n \\ y_i &= d_i - \sum_{\ell=i+1}^{n} p_{i,\ell} y_\ell . \end{aligned}\right\} \qquad (3.9.2)$$

for i=n-1(-1)s+2(m-1)

for i=s+2m-3(-1)s

$$y_i = d_i - \Big(\sum_{\ell=i+1}^{n} p_{i,\ell} y_\ell + \sum_{\ell=1}^{s-1} p_{i,\ell} y_\ell \Big). \qquad (3.9.3)$$

b) The outward substitution process for the solution of

$$\underline{Qx} = \underline{y} \ ,$$

is

$$x_s = y_s / q_{s,s} \ , \qquad (3.9.4)$$

followed by the forward substitution process from the centre given by,

for i=s+1(1)s+2m-3

$$x_i = (y_i - \sum_{\ell=s}^{i-1} q_{i,\ell} x_\ell)/q_{i,i} \ , \qquad (3.9.5)$$

for i=s+2(m-1)(1)n

$$x_i = (y_i - \sum_{\ell=i-2(m-1)}^{i-1} q_{i,\ell} x_\ell)/q_{i,i} \ , \qquad (3.9.6)$$

and the backward substitution from the centre,

for i=s-1(-1)1,

$$x_i = (y_i - \sum_{\ell=i+1}^{i+2(m-1)} q_{i,\ell} x_\ell)/q_{i,i} \ . \qquad (3.9.7)$$

During the inward substitution process, steps (3.9.1) and (3.9.2) are performed in parallel, and on completion are followed by step (3.9.3). Then the outward substitution process commences with steps (3.9.4) and (3.9.5) which must be completed before steps (3.9.6) and (3.9.7) are performed in parallel. Once again, in order to take full advantage of the accuracy of the method, double precision

accumulation of inner products must be used.

As with the tridiagonal algorithm, the number of arithmetic operations is dominated by the n^2 term which is due to the inward substitution stage, so we shall combine the substitution stage with the factorisation stage. This gives, in terms of arithmetic operations, for the sequential algorithm:

$$T_1 = m(8m-7)n-(m-1)(68m^2-55m+12)/6,$$

and for the parallel algorithm:

$$T_2 = \begin{cases} m(8m-7)n/2+(40m^3-102m^2+83m-18)/6, & \text{for n is odd} \\ m(8m-7)n/2+(m-1)(20m^2-43m+9)/3, & \text{for n is even,} \end{cases}$$

giving,

$$S_2 = \begin{cases} 2-(148m^3-327m^2+233m-48)/6.T_2, & \text{for n is odd} \\ 2-(m-1)(148m^2-227m+48)/6.T_2, & \text{for n is even.} \end{cases}$$

If speed-up is considered, it is clear that it is desirable for n to be large with respect to m, (i.e., n>>m).

3.10 THE GENERALISED SYMMETRIC PARALLEL FACTORISATION METHOD

The parallel triangular factorisation method has been successfully generalised, and now we proceed to generalise the symmetric parallel factorisation method. The factorisation of A is performed such that $Q=P^T$.

Matrix P will be identical in form to the shaded area of (3.8.1) with the exception that the leading diagonal will consist of entries $p_{i,i}$ (i=1(1)n) instead of unity values and

$$s = \begin{cases} \frac{n-m+3}{2} & \text{for (n-m) is odd} \\ \frac{n-m+2}{2} & \text{for (n-m) is even.} \end{cases} \qquad (3.10.1)$$

Then $Q=P^T$ (i.e. $q_{i,j}=p_{j,i}$) and the matrix product $P.P^T$ is defined as:

for $1 \leq i < s$

$$(PP^T)_{i,i+k} = (PP^T)_{i+k,i} = p_{i+k,i}p_{i,i} + \sum_{\ell=1}^{\alpha} p_{i+k,i-\ell}p_{i,i-\ell}, \quad \text{for } 0 \leq k \leq m-1$$

where the summation limit is defined to be,

$$\alpha = \begin{cases} i-1, & \text{for } i < (m-k) \\ m-k-1, & \text{for } i \geq (m-k), \end{cases} \qquad (3.10.2)$$

for $n \geq i \geq s+m-1$

$$(PP^T)_{i,i-k} = (PP^T)_{i-k,i} = p_{i-k,i}p_{i,i} + \sum_{\ell=1}^{\beta} p_{i-k,i+\ell}p_{i,i+\ell}, \quad \text{for } 0 \leq k \leq (m-1)$$

where

$$\beta \quad \begin{cases} n-i, & \text{for } (n-i+1) < (m-k) \\ m-k-1, & \text{for } (n-i+1) \geq (m-k), \end{cases} \qquad (3.10.3)$$

for $s+m-1 > i \geq s$

$$(PP^T)_{i,i-k} = (PP^T)_{i-k,i} = p_{i-k,i}p_{i,i} + \sum_{\ell=1}^{\beta} p_{i-k,i+\ell}p_{i,i+\ell} + \sum_{\ell=i-s+1}^{m-1} p_{i-k,i-\ell}p_{i,i-\ell}, \quad \text{for } 0 \leq k < i-s \qquad (3.10.4)$$

and β is defined in (3.10.3).

On equating the matrix PP^T with the matrix A, we have the following formulae for determining the elements of P and P^T. These are:

for $i = 1(1)s-1$

$$p_{i,i} = \sqrt{(a_{i,i} - \sum_{\ell=1}^{\alpha} p_{i,i-\ell}^2)}$$

$$p_{i+k,i} = (a_{i+k,i} - \sum_{\ell=1}^{\alpha} p_{i+k,i-\ell}p_{i,i-\ell})/p_{i,i}, \quad \text{for } k = 1(1)m-1 \qquad (3.10.5)$$

where α is defined in (3.10.2);

for $i = n(-1)s+m-1$

$$p_{i,i} = \sqrt{(a_{i,i} - \sum_{\ell=1}^{\beta} p_{i,i+\ell}^2)}$$

$$p_{i-k,i} = (a_{i-k,i} - \sum_{\ell=1}^{\beta} p_{i-k,i+\ell}p_{i,i+\ell})/p_{i,i}, \quad \text{for } k = 1(1)m-1 \qquad (3.10.6)$$

where β is defined in (3.10.3) and,

for i=s+m-2(-1)s

$$\left.\begin{aligned} p_{i,i} &= \sqrt{(a_{i,i} - (\sum_{\ell=1}^{\beta} p_{i,i+\ell}^2 + \sum_{\ell=i-s+1}^{m-1} p_{i,i-\ell}^2))} \\ p_{i-k,i} &= (a_{i-k,i} - (\sum_{\ell=1}^{\beta} p_{i-k,i+\ell}p_{i,i+\ell} + \sum_{\ell=i-s+1}^{m-1} p_{i-k,i-\ell}p_{i,i-\ell}))/p_{i,i} \\ &\qquad\qquad \text{for } k=1(1)i-s, \end{aligned}\right\}$$

where β is defined in (3.10.3). (3.10.7)

The solution of $P\underline{y}=\underline{d}$ for $\underline{y}$ is now given algorithmically as:

$$\left.\begin{aligned} y_1 &= \frac{d_1}{p_{11}} \quad , \\ \text{for } i=2(1)s-1 & \\ y_i &= (d_i - \sum_{\ell=1}^{\alpha} p_{i,i-\ell}y_{i-\ell})/p_{i,i} \\ \alpha &= \begin{cases} i-1 & \text{for } i<m \\ m-1 & \text{for } m\geq i \ , \end{cases} \end{aligned}\right\} \quad (3.10.8)$$

and

$$\left.\begin{aligned} y_n &= \frac{d_n}{p_{n,n}} \\ \text{for } i=(n-1)(-1)s+m-1 & \\ y_i &= (d_i - \sum_{\ell=1}^{\beta} p_{i,i+\ell}y_{i+\ell})/p_{i,i} \\ \beta &= \begin{cases} n-i & \text{for } n-i+1<m \\ m-1 & \text{for } n-i+1\geq m \end{cases} \end{aligned}\right\} \quad (3.10.9)$$

whilst for the interference area,

for i=s+m-2(-1)s

$$y_i = (d_i - (\sum_{\ell=1}^{\beta} p_{i,i+\ell}y_{i+\ell} + \sum_{\ell=i-s+1}^{m-1} p_{i,i-\ell}y_{i-\ell})/p_{i,i} \quad (3.10.10)$$

where β is defined in (3.10.9).

Finally, the solution of $Q\underline{x}=\underline{y}$ for $\underline{x}$ is now,

$$\left.\begin{aligned} x_s &= \frac{y_s}{p_{s,s}} \\ \text{for } i=s+1(1)s+m-2 & \\ x_i &= (y_i - \sum_{\ell=s}^{i+1} p_{\ell,i}x_\ell)/p_{i,i} \end{aligned}\right\} \quad (3.10.11)$$

for i=s+m-1(1)n

$$x_i = (y_i - \sum_{\ell=i-m+1}^{i-1} p_{\ell,i}x_\ell)/p_{i,i} \quad (3.10.12)$$

and <u>for i=s-1(-1)1</u>

$$x_i = (y_i - \sum_{\ell=i+1}^{i+m-1} p_{\ell,i} x_\ell)/p_{i,i} \quad . \tag{3.10.13}$$

The steps in the forward and backward substitution phases will be completed in the same order as described previously for the parallel factorisation (PQ) method.

Finally, we have that the number of arithmetic operations is:

$$T_2 = \begin{cases} (m^2+4m-2)n/2+(m-1)(m^2+7m-6)/6, & \text{for } (n-m) \text{ is odd} \\ (m^2+4m-2)n/2+m(m^2+9m-1)/6, & \text{for } (n-m) \text{ is even} \end{cases}$$

giving

$$S_2 = \begin{cases} 2-(m-1)(2m^2+9m-4)/2.T_2 \ , & \text{for } (n-m) \text{ is odd} \\ 2-m(2m^2+9m-5)/2.T_2 \ , & \text{for } (n-m) \text{ is even} \end{cases}$$

3.11 INHERENT PARALLELISM

The parallelism in the generalised factorisation method is basically the same as that for the method for tridiagonal systems. If we consider figures (3.1) and (3.3), then the corresponding diagrams for the generalised methods will be essentially the same. The diagram for the Generalised Parallel Triangular Factorisation method is:

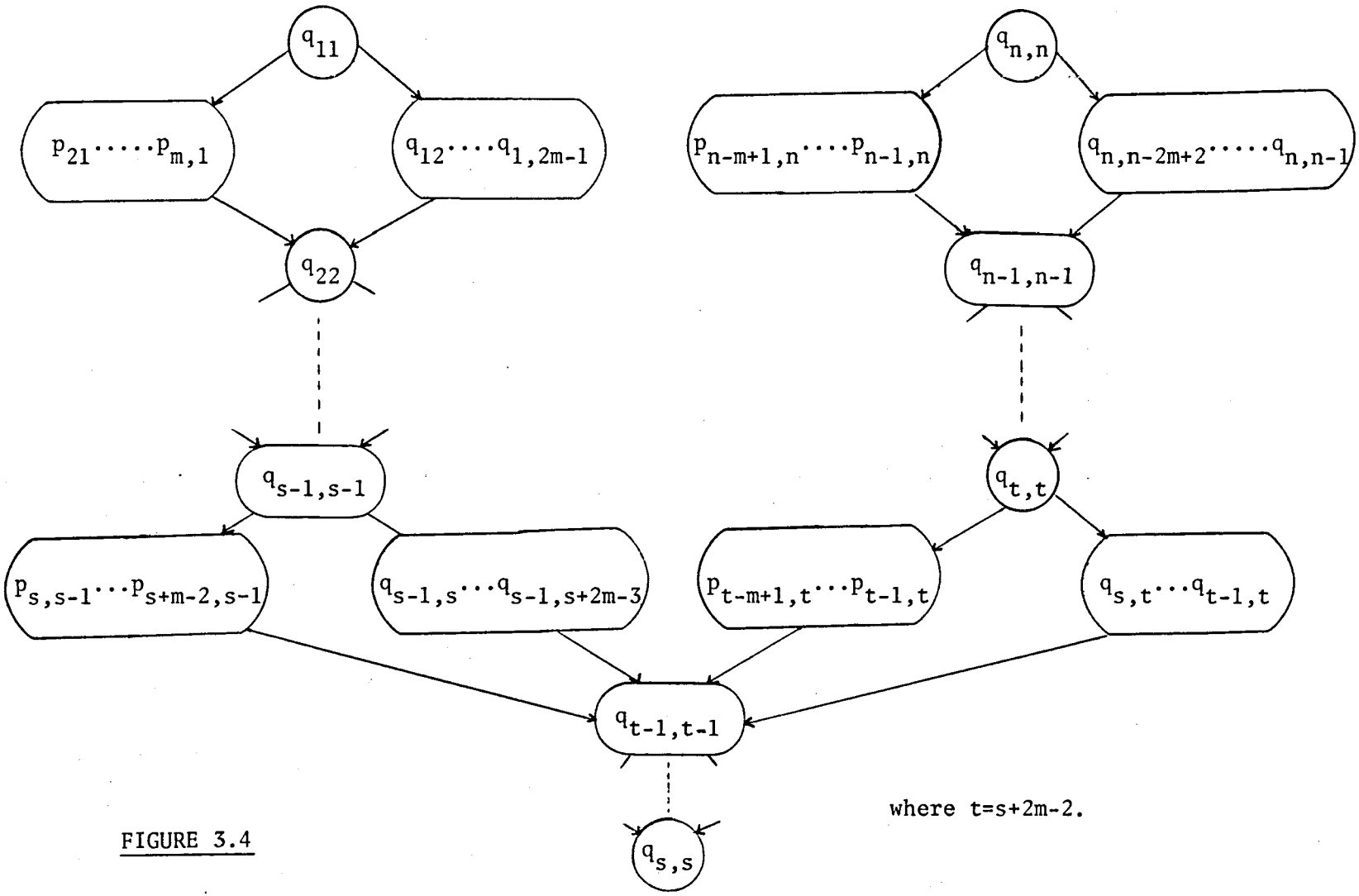


FIGURE 3.4

where t=s+2m-2.

It is obvious that a maximum of 6(m-1) processors may be used concurrently but the most suitable number of processors would be 2, because a larger number may greatly increase inefficiency. This is due to the need for synchronization within each of the two parallel branches which can only be achieved by a large number of forks and joins. In addition to this, the larger number of processors cannot be fully utilised in the critical regions of the algorithm.

Clearly the main disadvantage with these methods is the interaction area at the centre of the matrix $((a_{ij})$ i=s(1)s+2m-3, j=s(1)s+2m-3 in the case of the parallel factorisation method). As m increases, this area gradually fills the whole of matrix A and the method is reduced to the standard LU type factorisation. It is this area that reduces the effectiveness of the algorithms and by examining the speed-ups we may conclude that it is desirable for n to be large with respect to m, i.e., A is a narrow banded matrix.

3.12 ERROR ANALYSIS OF THE GENERALISED PARALLEL FACTORISATION METHOD

The following error analysis of the parallel factorisation method is an extension to the work pioneered by Wilkinson [1965]. Prior to the introduction of the complex analysis, we first consider some basic results.

For t-digit binary floating point computation and assuming that our computer has a double precision accumulator, then we can state the following definitions:

a number x is said to be rounded to t digits $x^{(t)}$ if

$$|\varepsilon| = |x-x^{(t)}| \leq \frac{1}{2}.2^{-t} \quad , \tag{3.12.1}$$

and for simple arithmetic operations we have

$$fl(x*y) = (x*y)(1+\varepsilon) \quad , \quad |\varepsilon| \leq 2^{-t} \quad , \tag{3.12.2}$$

where $fl(\ )$ indicates single precision (t-digits) and the operation * is +,-,× or ÷. Also we have,

$$fl_2(x*y) = (x+y)(1+\varepsilon) \ , \ |\varepsilon| \leq \frac{3}{2}\, 2^{-2t} \ , \tag{3.12.3}$$

where $fl_2(\ )$ indicates double precision (2t-digits) and the operation * is as before.

We are particularly interested in the error accumulation of double precision evaluation of inner products.

If we let

$$s_n = fl_2(x_1y_1+x_2y_2+\ldots+x_ny_n) \ , \tag{3.12.4}$$

where $x_1,x_2,\ldots,x_n$ and $y_1,y_2,\ldots,y_n$ are single precision numbers, the sum is accumulated in double precision in the accumulator and then rounded to a single precision number.

Let us denote,

$$\left.\begin{aligned} s_r &= fl_2(x_1y_1+x_2y_2+\ldots+x_ry_r) \\ \text{and} \qquad t_r &= fl_2(x_ry_r) \quad . \end{aligned}\right\} \tag{3.12.5}$$

Then, by developing the inner product recursively we can write,

$$\left.\begin{aligned} s_1 &= t_1 \ , \\ \text{and} \qquad s_r &= fl_2(s_{r-1}+t_r), \quad r=2,3,\ldots,n. \end{aligned}\right\} \tag{3.12.6}$$

Now, at each step of the recursion we have,

$$\left.\begin{aligned} t_r &= x_ry_r(1+\rho_r), \qquad |\rho_r| \leq \frac{3}{2}\, 2^{-2t} \ , \\ s_r &= (s_{r-1}+t_r)(1+\eta_r), \quad |\eta_r| \leq \frac{3}{2}\, 2^{-2t} \end{aligned}\right\} \tag{3.12.7}$$

and hence finally

$$s_n = x_1y_1(1+\varepsilon_1)+x_2y_2(1+\varepsilon_2)+\ldots+x_ny_n(1+\varepsilon_n) \tag{3.12.8}$$

where

$$\left.\begin{aligned} 1+\varepsilon_1 &= (1+\rho_1)(1+\eta_2)(1+\eta_3)\ldots(1+\eta_n) \ , \\ \text{and} \qquad 1+\varepsilon_r &= (1+\rho_r)(1+\eta_r)(1+\eta_{r+1})\ldots(1+\eta_n), \\ & \qquad\qquad\qquad\qquad\qquad r=2,3,\ldots n. \end{aligned}\right\} \tag{3.12.9}$$

Using the result of (3.12.3) we have

$$(1-\tfrac{3}{2}\,2^{-2t})^{n} \lesssim 1+\varepsilon_1 \lesssim (1+\tfrac{3}{2}\,2^{-2t})^{n}$$

and
$$(1-\tfrac{3}{2}\,2^{-2t})^{n-r+2} \lesssim 1+\varepsilon_r \lesssim (1+\tfrac{3}{2}\,2^{-2t})^{n-r+2} \quad . \qquad (3.12.10)$$

Rounding to t-digits we finally have

$$s_n = (x_1y_1(1+\varepsilon_1)+\ldots.x_ny_n(1+\varepsilon_n))(1+\varepsilon) \quad , \qquad (3.12.11)$$

where the ε_i are defined in (3.12.10) and we have for $(1+\varepsilon)$,

$$1-2^{-t} \lesssim 1+\varepsilon \lesssim 1+2^{-t} \quad . \qquad (3.12.12)$$

In order to simplify the bounds (3.12.10), it is reasonable to assume that since r is normally much smaller than 2^{2t} ,

$$\tfrac{3}{2}r\,2^{-2t} < 0.1 \quad . \qquad (3.12.13)$$

Then we have the result

$$(1+\tfrac{3}{2}\,2^{-2t})^{r} < 1+\tfrac{3}{2}r(1.06)2^{-2t} \quad , \qquad (3.12.14)$$

and introducing t_1 such that,

$$(1.06)2^{-2t} = 2^{-2t_1} \quad ,$$

i.e.,
$$2t_1 = 2t-\log_2(1.06) \quad , \qquad (3.12.15)$$

we can write the bounds for (3.12.11) as,

$$|\varepsilon| < 2^{-t} \quad ,$$
$$|\varepsilon_1| < \tfrac{3}{2}n.2^{-2t_1}$$

and
$$|\varepsilon_r| < \tfrac{3}{2}(n-r+2)2^{-2t_1} \quad . \qquad (3.12.16)$$

The remaining result that is required for the analysis that follows, concerns the division, before rounding, of an inner product accumulated in double precision, and is

$$fl_2((x_1y_1+x_2y_2+\ldots x_ny_n)/z) = \frac{x_1y_1(1+\varepsilon_1)+\ldots x_ny_n(1+\varepsilon_n)}{z/(1+\varepsilon)} \qquad (3.12.17)$$

where ε and ε_i are as defined in (3.12.16).

We commence the analysis of the method by considering the sensitivity of the solution $\underline{x}$ to perturbations in $\underline{d}$ and A. If (x+h) is the computed solution when A and $\underline{d}$ have been perturbed then

$$(A+F)(x+h) = (d+k) \quad . \tag{3.12.18}$$

If we subtract (3.1.1) we have

$$(A+F)h = k-Fx \tag{3.12.19}$$

$$\rightarrow \quad A(I+A^{-1}F)h = k-Fx$$

$$\rightarrow \quad h = (I+A^{-1}F)^{-1}A^{-1}(k-Fx)$$

$$\rightarrow \quad ||h|| \leq ||(I+A^{-1}F)^{-1}||.||A^{-1}||.||k-Fx|| \quad . \tag{3.12.20}$$

where $||A||$ denotes the norm of matrix A and can be defined by one of the following expressions,

$$||A||_1 = \max_j \sum_i |a_{i,j}| \quad ,$$

$$||A||_\infty = \max_i \sum_j |a_{i,j}| \quad ,$$

$$||A||_2 = (\text{maximum eigenvalue of } A^H A)^{\frac{1}{2}}$$

and

$$||A||_E = \left(\sum_i \sum_j |a_{i,j}|^2\right)^{\frac{1}{2}},$$

where A^H is the complex conjugate transpose A.

Assuming that

$$||A^{-1}F|| < 1 \quad , \tag{3.12.21}$$

then $(I+A^{-1}F)$ is non-singular and thus

$$\frac{1}{1+||A^{-1}F||} \leq ||I+A^{-1}F|| \leq \frac{1}{1-||A^{-1}F||} \quad , \tag{3.12.22}$$

and so

$$||h|| \leq \frac{1}{1-||A^{-1}F||} .||A^{-1}||.(||k||+||F||.||x||) \quad . \tag{3.12.23}$$

The relative error is of more interest and so

$$\frac{||h||}{||x||} \leq ||A||\ ||A^{-1}||\left\{\frac{||k||}{||d||} + \frac{||F||}{||A||}\right\} \frac{1}{1-||A^{-1}||.||F||} \quad . \tag{3.12.24}$$

If only the perturbations in $\underline{d}$ are considered, F=0 and

$$\frac{||h||}{||x||} = ||A||.||A^{-1}|| \frac{||k||}{||d||} \quad . \tag{3.12.25}$$

Alternatively, by only considering perturbations in A, k=0 and

$$\frac{||h||}{||x||} \leq ||A||\ ||A^{-1}||.\ \frac{||F||}{||A||}\ .\ \frac{1}{1-||A^{-1}||\ ||F||}$$

or

$$\frac{||h||}{||x||} \leq ||A||\ ||A^{-1}||.\ \frac{||F||}{||A||}\ .\ \frac{1}{1\ -||A||\,||A^{-1}||\frac{||F||}{||A||}} \qquad (3.12.26)$$

In (3.12.25) we have a bound for $||h||/||x||$ in terms of $||k||/||d||$ and in (3.12.26) in terms of $||F||/||A||$. For both bounds we see that the decisive quantity is $||A||.||A^{-1}||$ which is known as the condition number.

Now let us consider the decomposition of the matrix A into the product P.Q.

Excluding rounding errors we know that the augmented matrix $(A\,|\,d)$ with its rows permuted is equal to the augmented matrix $P(Q\,|\,y)$. We require a bound for

$$\{P(Q\,|\,y) - (A\,|\,d)\} \qquad (3.12.27)$$

where $(A\,|\,d)$ represents the permuted matrix.

In general, it can be expected that the maximum $|q_{i,j}|$ rarely exceeds the maximum $|a_{i,j}|$ by any appreciable factor and, in fact, when A is ill-conditioned the $|a_{i,j}|$ will usually be greater than $|q_{i,j}|$.

If we scale A we will have some control over the size of elements where necessary; so scale A such that all $|a_{i,j}|<\frac{1}{2}$. Then, by examining the quantities R_t and $q_{i,t}$ as they are accumulated, if an inner product exceeds $\frac{1}{2}$ in absolute value, we divide either R_t or $q_{i,t}$ and the complete row t or i of A and d by two. It is expected however that the necessity for such a division is rare.

Thus, assuming no divisions are necessary we have

for $1 \leq i < s$

$$p_{i+k,i} = \frac{R_{i+k}}{q_{i,i}} + \varepsilon_{i+k,i} \quad , |\varepsilon_{i+k,i}| \leq \frac{1}{2}.2^{-t} \tag{3.12.28}$$

where $p_{i+k,i}$, s_{i+k} and $q_{i,i}$ refer to computed values. From (3.8.7) we may rewrite (3.12.28) as

$$a_{i+k,i} = \sum_{\ell=\alpha}^{i-1} p_{i+k,\ell} \cdot q_{\ell,i} + p_{i+k,i} \cdot q_{i,i} + q_{i,i} \cdot \varepsilon_{i+k,i} \ . \tag{3.12.29}$$

Also we have for $q_{i,i+k}$ and y_i

$$\left.\begin{aligned} q_{i,i+k} &= a_{i,i+k} - \sum_{\ell=\alpha}^{i-1} p_{i,\ell} q_{\ell,i+k} + \varepsilon_{i,i+k}, \quad |\varepsilon_{i,i+k}| \leq \tfrac{1}{2}.2^{-t} \\ \text{and} \quad y_i &= d_i - \sum_{\ell=1}^{i-1} p_{i,\ell} y_\ell + \varepsilon_i \ , \quad |\varepsilon_i| \leq \tfrac{1}{2}.2^{-t} \ . \end{aligned}\right\} \tag{3.12.30}$$

Similarly we have the following results:-

for $n \geq i \geq s+2(m-1)$

$$\left.\begin{aligned} a_{i-k,i} &= \sum_{\ell=i+1}^{\beta} p_{i-k,\ell} q_{\ell,i} + p_{i-k,i} q_{i,i} + q_{i,i} \varepsilon_{i-k,i} \\ &\qquad |\varepsilon_{i-k,i}| \leq \tfrac{1}{2}.2^{-t} \\ q_{i,i-k} &= a_{i,i-k} - \sum_{\ell=i+1}^{\beta} p_{i,\ell} q_{\ell,i-k} + \varepsilon_{i,i-k} \ , \\ &\qquad |\varepsilon_{i,i-k}| \leq \tfrac{1}{2}.2^{-t} \\ y_i &= d_i - \sum_{\ell=i+1}^{n} p_{i,\ell} y_\ell + \varepsilon_i \ , \quad |\varepsilon_i| \leq \tfrac{1}{2}.2^{-t} \end{aligned}\right\} \tag{3.12.31}$$

and finally,

for $s+2(m-1) > i \geq s$

$$\left.\begin{aligned} a_{i-k,i} &= \Big(\sum_{\ell=i+1}^{\beta} p_{i-k,\ell} q_{\ell,i} + \sum_{\ell=\alpha}^{s-1} p_{i-k,\ell} q_{\ell,i} \Big) \\ &\quad + p_{i-k,i} q_{i,i} + q_{i,i} \varepsilon_{i-k,i}, |\varepsilon_{i-k,i}| \leq \tfrac{1}{2}.2^{-t} \\ q_{i,i-k} &= a_{i,i-k} - \Big(\sum_{\ell=i+1}^{\beta} p_{i,\ell} q_{\ell,i-k} + \sum_{\ell=\alpha}^{s-1} p_{i,\ell} q_{\ell,i-k} \Big) + \varepsilon_{i,i-k} \\ &\qquad |\varepsilon_{i,i-k}| \leq \tfrac{1}{2}.2^{-t} \\ y_i &= d_i - \Big(\sum_{\ell=i+1}^{n} p_{i,\ell} y_\ell + \sum_{\ell=1}^{s-1} p_{i,\ell} y_\ell \Big) + \varepsilon_i, \ |\varepsilon_i| \leq \tfrac{1}{2}.2^{-t} \ . \end{aligned}\right\} \tag{3.12.32}$$

It may be shown that, by taking terms in P,Q and y to one side,

$$P(Q\,\vdots\,y) \equiv (A+F, d+k) \tag{3.12.33}$$

where

$$|f_{i,j}| \lesssim \left\{ \begin{array}{ll} \frac{1}{2}.2^{-t}, & s>i\lesssim j \quad \text{and} \quad s\lesssim i\gtrsim j \\ \frac{1}{2}|q_{i,i}|.2^{-t}, & s>j<i \text{ and } s<j>i \end{array} \right\} \tag{3.12.34}$$

and $|k_i| \lesssim \frac{1}{2}.2^{-t}$.

However, all $|f_{i,j}| \lesssim \frac{1}{2}.2^{-t}$, since we have assumed that $|q_{i,j}| \lesssim 1$. Obviously, for many of the elements $|f_{i,j}|$, this bound is pessimistic as some $|q_{i,i}|$ are considerably smaller than unity.

Finally, we must consider the two substitution stages defined by

$$P\underline{y} = \underline{d} \qquad \text{and} \qquad Q\underline{x} = \underline{y} \ .$$

The analysis of the solution of both sets of equations is similar and first we consider the solution of $Q\underline{x} = \underline{y}$.

Now <u>for i=s(1)s+2m-3</u>

if we assume that $x_s, x_{s+1}, \ldots, x_{i-1}$ have already been computed, then

$$\begin{aligned} x_i &= fl_2[-q_{i,s}x_s - q_{i,s+1}x_{s+1} \cdots - q_{i,i-1}x_{i-1} + y_i]/q_{i,i} \\ &\equiv [-q_{i,s}x_s(1+\varepsilon_s) - q_{i,s+1}x_{s+1}(1+\varepsilon_{s+1}) \cdots - q_{i,i-1}x_{i-1}(1+\varepsilon_{i-1}) \\ &\qquad + y_i(1+\varepsilon_i)] \times \frac{(1+\varepsilon)}{q_{i,i}} \end{aligned} \tag{3.12.35}$$

where $|\varepsilon_k| < \frac{3}{2}(i-k+2)2^{-2t_1}$, $|\varepsilon_i| \lesssim \frac{3}{2}.2^{-2t_1}$, $|\varepsilon| \lesssim 2^{-t}$ and by dividing the denominator and numerator by $(1+\varepsilon_i)$, we have

$$\begin{aligned} x_i = [&-q_{i,s}x_s(1+\eta_s) - q_{i,s+1}x_{s+1}(1+\eta_{s+1}) \cdots - q_{i,i-1}x_{i-1}(1+\eta_{i-1}) \\ &+ y_i]/q_{i,i}(1+\eta) \end{aligned} \tag{3.12.36}$$

where certainly,

$$|\eta_k| \lesssim \frac{3}{2}(i-k+3)2^{-2t_1} \quad \text{and} \quad |\eta| < 2^{-t}(1.00001) \ .$$

By a rearrangement of (3.12.36), we may write,

$$q_{i,s}x_s(1+\eta_s)+q_{i,s+1}x_{s+1}(1+\eta_{s+1})+\ldots+q_{i,i-1}x_{i-1}(1+\eta_{i-1})$$
$$+q_{i,i}x_i(1+\eta) = y_i \ . \qquad (3.12.37)$$

Similarly, we have

<u>for i=s+2(m-1)(1)n</u>

$$q_{i,\ell}x_\ell(1+\eta_\ell)+q_{i,\ell+1}x_{\ell+1}(1+\eta_{\ell+1})+\ldots q_{i,i-1}x_{i-1}(1+\eta_{i-1})$$
$$+q_{i,i}x_i(1+\eta) = y_i \ , \qquad (3.12.38)$$

where $\ell=i-2(m-1)$ and $|\eta_k|\leq\frac{3}{2}(i-k+3)2^{-2t_1}$, $|\eta|<2^{-t}(1.00001)$

and <u>for i=s-1(-1)1</u>

$$q_{i,i+1}x_{i+1}(1+\eta_{i+1})+q_{i,i+2}x_{i+2}(1+\eta_{i+2})+\ldots+q_{i,\ell}x_\ell(1+\eta_\ell)$$
$$+q_{i,i}x_i(1+\eta) = y_i \ , \qquad (3.12.39)$$

where $\ell=i+2(m-1)$ and $|\eta_k|\leq\frac{3}{2}(\ell-k+4)$, $|\eta|<2^{-t}(1.00001)$.

It is now obvious that x_i, the computed solution, will satisfy exactly the equation,

$$(Q+\delta Q)\underline{x} = \underline{y} \ , \qquad (3.12.40)$$

where δQ is bounded by

$$|\delta Q|\leq 2^{-t}(1.00001)\begin{bmatrix}|q_{11}| & & & \\ & |q_{22}| & & \\ & & \ddots & \\ & & & |q_{n,n}|\end{bmatrix} +$$

$$\frac{3}{2}\cdot 2^{-2t_1}\begin{bmatrix}0 & (2m+1)|q_{12}| & \ldots & 4|q_{1,2m-1}| & & & \\ & 0 & \ddots & & \ddots & & \\ & & 0 & (2m+1)|q_{s-1,s}| & \ldots & 4|q_{s-1,s+2m-3}| & \\ & & & 0 & & & \\ & & & 4|q_{s+1,s}| & 0 & \ddots & \\ & & & \vdots & & \ddots & \\ & & & (2m+1)|q_{s+2m-2,s}| & & & \\ & & & & \ddots & & \\ & & & & & (2m+1)|q_{n,n-2m+2}| & \ldots 4|q_{n,n-1}| \quad 0\end{bmatrix} \qquad (3.12.41)$$

For the 1,2 and ∞ norms we have

$$||\delta Q|| \lesssim 2^{-t}(1.00001)g + 3.2^{-2t_1}(2m^2+3m-5)g$$

where g is max $|q_{i,j}|$. Now should $g \lesssim 1$ then

$$||\delta Q|| \lesssim 2^{-t}(1.00001) + 3.2^{-2t_1}(2m^2+3m-5) \quad . \tag{3.12.42}$$

If $m^2.2^{-t} << 1$, the second term is negligible.

Now considering the residual vector

$$(\underline{y} - Q\underline{x}) = \delta Q\underline{x} \quad , \tag{3.12.43}$$

then

$$||\underline{y} - Q\underline{x}|| \lesssim ||\delta Q||.||\underline{x}|| \quad . \tag{3.12.44}$$

Now if x_e, the exact solution, when rounded to t figures gives $\bar{x}$, then we may have

$$\bar{x} = x_e + c \quad . \tag{3.12.45}$$

It is obvious that

$$||c||_\infty \lesssim 2^{-t}||x_e||_\infty \quad , \tag{3.12.46}$$

and hence,

$$\underline{y} - Q\bar{x} = \underline{y} - Q(x_e+c) = -Qc \ ,$$

thus

$$||y-Q\bar{x}||_\infty = ||Qc||_\infty \lesssim ||Q||_\infty ||c||_\infty \lesssim (2m-1)2^{-t}||x_e||_\infty . \tag{3.12.47}$$

Since it would be easy to devise an example that achieves this bound, then following Wilkinson [1965] we can say <u>that we may expect the residuals corresponding to the computed solution of the triangular set of equations to be smaller than those corresponding to the correctly rounded solution</u>.

The analysis for the solution of $P\underline{y}=\underline{d}$ is very similar, but the diagonal elements of P are unity and so there will be no divisions involved in this stage.

We have that the computed vector $\underline{y}$ will satisfy

$$(P+\delta P)\underline{y} = \underline{d} \quad , \tag{3.12.48}$$

where δP is bounded by

$$|\delta P| \leq 2^{-t}(1.00001)\begin{bmatrix} 1 & & \\ & 1 & \\ & & \ddots \\ & & & 1 \end{bmatrix} + \qquad (3.12.49)$$

$$\frac{3}{2}.2^{-2t_1}\begin{bmatrix}
0 & & & & & & & & & \\
4|p_{21}| & 0 & & & & & & & & \\
5|p_{31}| & 4|p_{32}| & 0 & & & & & & & \\
\vdots & & \ddots & \ddots & & & & & & \\
(s+2)|p_{s,1}| & \cdots & 4|p_{s,s-1}| & 0 & (n+2)|p_{s,s+1}| & \cdots & \cdots & \cdots & (s+3)|p_{s,n}| \\
\vdots & & \vdots & & 0 & \ddots & & & \vdots \\
(s+2)|p_{r,1}| & \cdots & 4|p_{r,s-1}| & & & 0 & (n-2m+5)|p_{r,r+1}| & & (s+3)|p_{r,n}| \\
 & & & & & & 0 & (n-s-2m+5)|p_{r+1,r+2}| & 4|p_{r+1,n}| \\
 & & & & & & & \ddots & \vdots \\
 & & & & & & & & 4|p_{n-1,n}| \\
 & & & & & & & & 0
\end{bmatrix}$$

where $r=s+2m-3$.

The elements of P are bounded by unity and so

$$||\delta P|| < 2^{-t}(1.00001) + \frac{3}{4}.2^{-2t_1}(n+3)^2 \qquad (3.12.50)$$

for the 1,2 and ∞ norms. Again, if $n^2 2^{-t} \ll 1$, the second term is negligible.

As with the solution of $Q\underline{x}=\underline{y}$ we have,

$$||\underline{d} - P\underline{y}|| \lesssim ||\delta P||\,||\underline{y}|| \qquad (3.12.51)$$

and for the exact solution, y_e

$$||\underline{d} - P\bar{\underline{y}}||_\infty \lesssim n2^{-t}||y_e||_\infty \qquad (3.12.52)$$

giving that we can expect the residuals corresponding to the computed solution to be smaller than those of the correctly rounded solution.

If we return to the original problem to ascertain the errors in the solution of

$$A\underline{x} = \underline{d} \quad ,$$

we observe that A is factorised into the matrices P and Q such that

$$P.Q = A+F \quad , \qquad (3.12.53)$$

where F is the same matrix as in (3.12.33).

The solution $\underline{x}$ is obtained by solving the two sets of equations $P\underline{y}=\underline{d}$ and $Q\underline{x}=\underline{y}$ and the computed solutions $\underline{x}$ and $\underline{y}$ satisfy exactly the equations,

$$(P+\delta P)\underline{y} = \underline{d} \quad \text{and} \quad (Q+\delta Q)\underline{x} = \underline{y} \quad ,$$

where the bounds for δP and δQ are given in (3.12.49) and (3.12.41).

Hence $\underline{x}$ satisfies

$$(P+\delta P)(Q+\delta Q)\underline{x} = \underline{d} \quad , \qquad (3.12.54)$$

that is,

$$(A+G)\underline{x} = \underline{d} \quad , \qquad (3.12.55)$$

where

$$G = F+\delta P.Q + P.\delta Q + \delta P.\delta Q \quad . \qquad (3.12.56)$$

Now assuming partial pivoting has been used and floating point computation with double precision accumulation of inner products, and also that $|a_{i,j}| \lesssim 1$ and the $|u_{i,j}|$ have remained less than unity then,

$$||G||_\infty \leq (2m-1)\frac{1}{2}2^{-t} + (2m-1)2^{-t}(1.00001) + n.2^{-t}(1.00001) + 0(mn^2 2^{-2t_1}) \tag{3.12.57}$$

If $n2^{-t}$ is appreciably less than unity then we have that

$$||G||_\infty \leq (1.1n+3.1m)2^{-t} \quad \text{(approximately)} .$$

Since $m<n$ it is clear that the majority of the upper bound arises from the solution of the two sets of equations.

In terms of residuals, from (3.12.55) we have

$$\underline{r} = \underline{d} - A\underline{x} = G\underline{x}$$

and thus $$||\underline{r}||_\infty \leq ||G||_\infty \; ||\underline{x}||_\infty \leq (1.1n+3.1m)2^{-t}||\underline{x}||_\infty . \tag{3.12.58}$$

This residual bound is in terms of the size of the computed solution and not its accuracy.

Wilkinson [1965] also demonstrates how to improve, iteratively, the computed solution by using iterative refinement which is defined as:

$$\left.\begin{aligned} &\underline{x}^{(0)} = 0, \qquad \underline{r}^{(0)} = \underline{d} \quad , \\ &PQ\underline{c}^{(k)} = \underline{r}^{(k)} \; , \; \underline{x}^{(k+1)} = \underline{x}^{(k)} + \underline{c}^{(k)} \; , \; \underline{r}^{(k+1)} = \underline{d} - A\underline{x}^{(k+1)} \end{aligned}\right\} \tag{3.12.59}$$

where $PQ=A+F$ and the $\underline{x}^{(k)}$ are a sequence of approximations to the true solution $\underline{x}$. If performed without rounding errors this process yields

$$\begin{aligned} \underline{x}^{(k+1)} &= \underline{x}^{(k)} + (A+F)^{-1}(\underline{b} - A\underline{x}^{(k)}) \\ &= \underline{x}^{(k)} + (A+F)^{-1}A(\underline{x} - \underline{x}^{(k)}) \end{aligned} \tag{3.12.60}$$

which, on subtracting $\underline{x}$ from both sides and rearranging, becomes

$$\begin{aligned} (\underline{x}^{(k+1)} - \underline{x}) &= [I - (A+F)^{-1}A](\underline{x}^{(k)} - \underline{x}) \\ &= [I - (A+F)^{-1}A]^k(\underline{x}^{(1)} - \underline{x}) \end{aligned} \tag{3.12.61}$$

If $$||I - (A+F)^{-1}A|| < 1 \tag{3.12.62}$$

then this is a sufficient condition for the convergence of the exact iterative process. This is satisfied if

$$||A^{-1}|| \; ||F|| < 1 \quad , \tag{3.12.63}$$

however, since

$$||F||_\infty \leq \frac{1}{2}(2m-1)2^{-t} \quad , \tag{3.12.64}$$

then the iterative process converges for

$$||A^{-1}||_\infty < 2^t/(2m-1) \quad . \tag{3.12.65}$$

3.13 EXAMPLES

Consider the following (10×10) linear systems where the right hand side vector $\underline{d}$ has been suitably chosen to make the solution vector possess unit elements.

a) if

$$A=\begin{bmatrix} 2 & -1.5 & & & & & & & & \\ -0.5 & 3 & -1.5 & & & & & & & \\ & -0.5 & 4 & -1.5 & & & & & & \\ & & -0.5 & 5 & -1.5 & & & & & \\ & & & -0.5 & 6 & -1.5 & & & & \\ & & & & -0.5 & 7 & -1.5 & & & \\ & & & & & -0.5 & 8 & -1.5 & & \\ & & & & & & -0.5 & 9 & -1.5 & \\ & & & & & & & -0.5 & 10 & -1.5 \\ & & & & & & & & -0.5 & 11 \end{bmatrix} \text{ and } \underline{d}=\begin{bmatrix} 0.5 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 10.5 \end{bmatrix}$$

then by the matrix factorisation process of section (3.3) without pivoting, we have the following results,

$$P = \begin{bmatrix} 1 & & & & & & & & & & \\ -0.25 & 1 & & & & & & & & & \\ & -0.19048 & 1 & & & & & & & & \\ & & -0.13462 & 1 & & & & & & & \\ & & & -0.10421 & 1 & -0.21723 & & & & & \\ & & & & & 1 & -0.18949 & & & & \\ & & & & & & 1 & -0.16808 & & & \\ & & & & 0 & & & 1 & -0.15103 & & \\ & & & & & & & & 1 & -0.13636 & \\ & & & & & & & & & 1 & \end{bmatrix}$$

and

$$Q = \begin{bmatrix} 2 & -1.5 & & & & & & & & & \\ 2.625 & -1.5 & & & & & & & & & \\ & 3.71429 & -1.5 & & & & & & & & \\ & & 4.79808 & -1.5 & & & & & & & \\ & & & 5.73507 & & & & & & & \\ & & & -0.5 & 6.90525 & & & & & & \\ & & & & -0.5 & 7.91596 & & & & & \\ & & & & & -0.5 & 8.92449 & & & & \\ & & & & 0 & & & -0.5 & 9.93182 & & \\ & & & & & & & & -0.5 & 11 & \end{bmatrix}$$

b) Similarly the square root factorisation method outlined in section (3.7) yields the component matrices P and P^T, for the symmetric system $A\underline{x}=\underline{d}$, and is given as follows:

$$A = \begin{bmatrix} 2 & -1 & & & & & & & \\ -1 & 3 & -1 & & & & & & \\ & -1 & 4 & -1 & & & & & \\ & & -1 & 5 & -1 & & & & \\ & & & -1 & 6 & -1 & & & \\ & & & & -1 & 7 & -1 & & \\ & & & & & -1 & 8 & -1 & \\ & & & & & & -1 & 9 & -1 \\ & & & & & & & -1 & 10 \end{bmatrix} \qquad \underline{d} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 10 \end{bmatrix}$$

$$P = \begin{bmatrix} 1.41421 & & & & & & & & \\ -0.70711 & 1.58114 & & & & & & & \\ & -0.63246 & 1.89737 & & & & & & \\ & & -0.52705 & 2.17307 & & & & & \\ & & & -0.46018 & 2.37515 & -0.38143 & & & \\ & & & & & 2.62168 & -0.35606 & & \\ & & & & & & 2.80846 & -0.33522 & \\ & & & & & & & 2.98313 & -0.31768 \\ & & & & & & & & 3.14781 & -0.30151 \\ & & & & & & & & & 3.31662 \end{bmatrix}$$

c) The third example illustrates the factorisation method of (3.3) incorporating the pivoting strategy. If

$$A = \begin{bmatrix} 3 & 4 & & & & & & \\ 2 & 1 & 5 & & & & & \\ & 3 & 4 & 2 & & & & \\ & & 6 & 4 & 3 & & & \\ & & & 4 & 3 & 5 & & \\ & & & & 1 & 4 & 2 & \\ & & & & & 2 & 1 & 3 \\ & & & & & & 2 & 4 \end{bmatrix} \quad \text{and} \quad d = \begin{bmatrix} 7 \\ 8 \\ 9 \\ 13 \\ 12 \\ 7 \\ 6 \\ 6 \end{bmatrix}$$

then the factors are:

$$P = \begin{bmatrix} 1 & & & & & & & \\ 0 & 1 & & & & & & \\ 0.66667 & -0.55556 & 1 & & & & & \\ 0 & 0 & 0 & 1 & -.51667 & 0.6 & -0.25 & 0.75 \\ 0 & 0 & 0.83077 & 0 & 1 & 0 & 0 & 0 \\ & & & & & 1 & 0 & 0 \\ & & & & & & 1 & 0 \\ & & & & & & & 1 \end{bmatrix}$$

and

$$Q = \begin{bmatrix} 3 & 4 & & & & & & \\ & 3 & 4 & 2 & & & & \\ & & 7.22222 & 1.11111 & & & & \\ & & & -0.81026 & & & & \\ & & & 3.07692 & 3 & & & \\ & & & 4 & 3 & 5 & & \\ & & & & 1 & 4 & 2 & \\ & & & & & & 2 & 4 \end{bmatrix}$$

N.B. The examples in this section have been evaluated on the ICL 1904S computer at Loughborough University and the results rounded to 5 decimal figures.

CHAPTER 4

THE SOLUTION OF TRIANGULAR SYSTEMS OF EQUATIONS

4.1 INTRODUCTION

In Chapter 3, the solution of a banded system of equations (3.1.1) was investigated. Another commonly occurring problem in numerical mathematics is the solution of the system of equations,

$$M\underline{y} = \underline{b} \quad , \tag{4.1.1}$$

where M is an (n×n) triangular matrix of the form (4.2.1) and $\underline{b}$ and $\underline{y}$ are (n×1) column vectors.

The sequential algorithm approach to the problem is a forward substitution process when M is lower triangular and a backward substitution process when M is upper triangular. As an example, the forward substitution process for the solution of the system of equations (4.1.1) is defined as,

$$y_i = (b_i - \sum_{j=1}^{i-1} m_{i,j}y_j)/m_{i,i}, \quad \text{for } i=1(1)n \ . \tag{4.1.2}$$

This algorithm is essentially sequential in that y_2 is dependent on the value of y_1,y_3 on the values of y_2 and y_1,y_4 on y_3,y_2 and y_1, etc., and this restricts the number of processors that can be used and speed-up that can be obtained by implementing it as a parallel algorithm. However, it is possible to substitute the value of y_1 into equations 2,3,4,...n simultaneously and then the value of y_2 into equations 3,4,5,...n simultaneously etc., and so it is not difficult to see that the simple idea of assigning one processor to each equation in system (4.1.1) yields the maximum speed-up for this strategy. Unfortunately, this simple algorithm is inefficient because the processors become idle as the algorithm progresses.

In the following study, various strategies for employing more than one processor to execute the substitution process efficiently are investigated. Other algorithms, such as that of Chen and Kuck [1975],

which is extremely fast but inefficient, are also considered and compared with the new strategies presented here by means of an index called the performance factor (4.2.6), which is a quantity that attempts to find a balance between speed-up and efficiency.

The essential difference between the two types of algorithm is that Chen and Kuck attempt to solve the problem in as short a time as possible, regardless of the number of processors that are required, which is frequently unrealistically large. The algorithms presented here, however, attempt to use a smaller number of processors efficiently. It is shown, in fact, that one of these methods, the Parallel Wave Front Method, has, in the majority of cases, the best performance.

4.2 THE SEQUENTIAL SUBSTITUTION PROCESS

Matrix M is an (n×n) lower triangular matrix of the form:

$$M = \begin{bmatrix} m_{11} & & & \\ m_{21} & m_{22} & & 0 \\ \vdots & \vdots & \ddots & \\ m_{n1} & m_{n2} & \cdots & m_{nn} \end{bmatrix} \qquad (4.2.1)$$

M could also be upper triangular in form, in which case the analysis would be similar.

In order to simplify the system and permit direct comparison with the algorithm of Chen and Kuck [1975], we shall perform the following transformation of the general system (4.1.1) to the form

$$\underline{y} = \underline{d} + A\underline{y} \qquad (4.2.2)$$

where

$$A = \begin{bmatrix} 0 & & & & \\ a_{21} & 0 & & & \\ a_{31} & a_{32} & 0 & & 0 \\ \vdots & \vdots & \ddots & \ddots & \\ a_{n1} & a_{n2} & \cdots & a_{n,n-1} & 0 \end{bmatrix} \qquad (4.2.3)$$

and $\underline{d}$ is a column vector such that

$$\left.\begin{aligned} d_i &= b_i/m_{i,i} && \text{for } i=1,2,\ldots n \\ \text{and} \quad a_{i,j} &= -m_{i,j}/m_{i,i} && \text{for } j=1,2,\ldots i-1, \\ & && \quad i=2,3,\ldots n \end{aligned}\right\}. \qquad (4.2.4)$$

This new system may be solved by a forward substitution process described as follows:

<u>for i=1(1)n</u>

$$y_i = d_i + \sum_{j=1}^{i-1} a_{i,j} y_j \quad . \qquad (4.2.5)$$

Before we consider any parallel strategies we shall make the following assumptions and definitions. First of all we shall assume that each processor being identical works at the same speed and secondly, that each arithmetic operation requires the same amount of time called a unit step. Finally we define an algorithm step as one multiplication followed by an addition which is equal to two unit steps.

In Chapter 2 the quantities T_i, S_i and E_i were defined. We now introduce an additional quantity called the performance factor which is consistent with our previous definitions and is defined as

$$PF_i = E_i \times S_i \quad . \qquad (4.2.6)$$

By combining efficiency with speed-up we have an index that enables us to assess the optimum number of processors that may be used for the solution of the problem.

It is obvious that a uniprocessor will solve the system (4.2.2) sequentially in n(n-1) unit steps by the forward substitution process (4.2.5). So we can say that

$$T_1 = n(n-1) \text{ unit steps} . \tag{4.2.7}$$

With a computer that has p processors, we can perform p operations concurrently and therefore have a minimum time requirement for the solution of (4.2.2) of

$$\min(T_p) = \frac{n(n-1)}{p} \text{ unit steps} . \tag{4.2.8}$$

It is not easy to achieve this limit as we would have to organise the processors so that they were not left lying idle at any point during the processing period.

Using the forward substitution method it is clear that to complete the evaluation of y_n we require y_{n-1}. Similarly we require y_{n-2} to complete the evaluation of y_{n-1} and so on, so that any algorithm based on this method requires a minimum of 2(n-1) unit steps. From (4.2.8) it is obvious that a minimum of $\frac{1}{2}n$ processors are necessary to solve the problem in this time.

Suppose that there are (n-1) processors available, then we may assign one processor to each equation of the system (4.2.2). Obviously an increase in p, the number of processors, such that p>(n-1) is not beneficial as (n-1) is the maximum number of processors that may be used and the extra processors would only be redundant. Assuming that the y_i already contain the d_i, then by assigning one processor to each equation we have,

Clock count in unit steps

Processor	Time → 0	2	4	..... (2n-1)
1 →	$y_2 = y_2 + a_{21}y_1$			
2 →	$y_3 = y_3 + a_{31}y_1 + a_{32}y_3$			
⋮	⋮			
(n-1) →	$y_n = y_n + a_{n1}y_1 + a_{n2}y_2 + \ldots\ldots a_{n,n-1}y_{n-1}$.			

(4.2.9)

Only processor (n-1) will be occupied for the complete 2(n-1) unit steps, the other processors becoming idle as they complete the evaluation of the equations to which they are assigned.

This can be demonstrated more clearly in the following diagram:

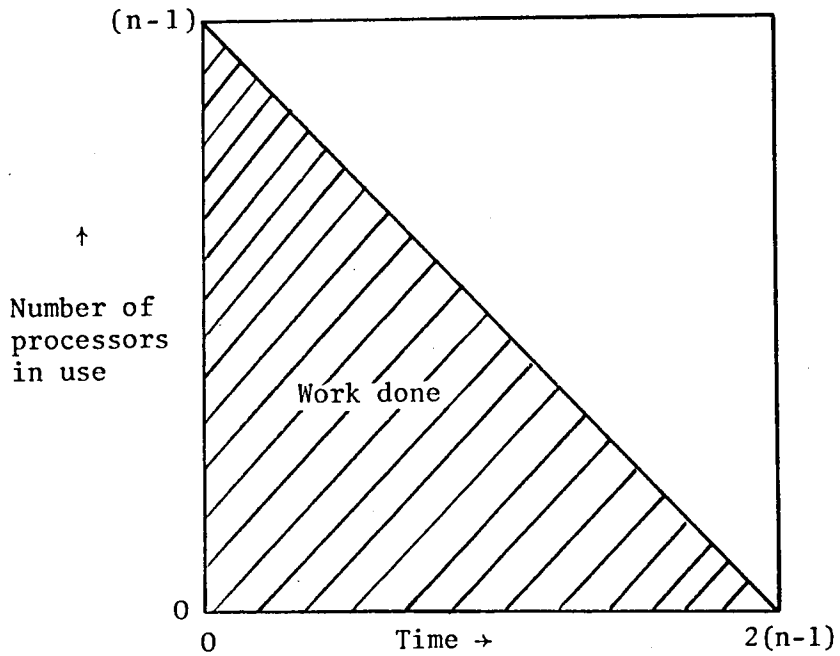


FIGURE 4.1

The complete area of the square represents the total capacity for work of the parallel computer. The shaded area is the used capacity, where processors are in use and the unshaded area the wasted capacity where processors are lying idle. Thus the system is being used inefficiently as half of its potential capacity for work is wasted.

Now let us investigate some new strategies that will improve the efficiency of the computer. These strategies fall into three categories depending on the number of processors available.

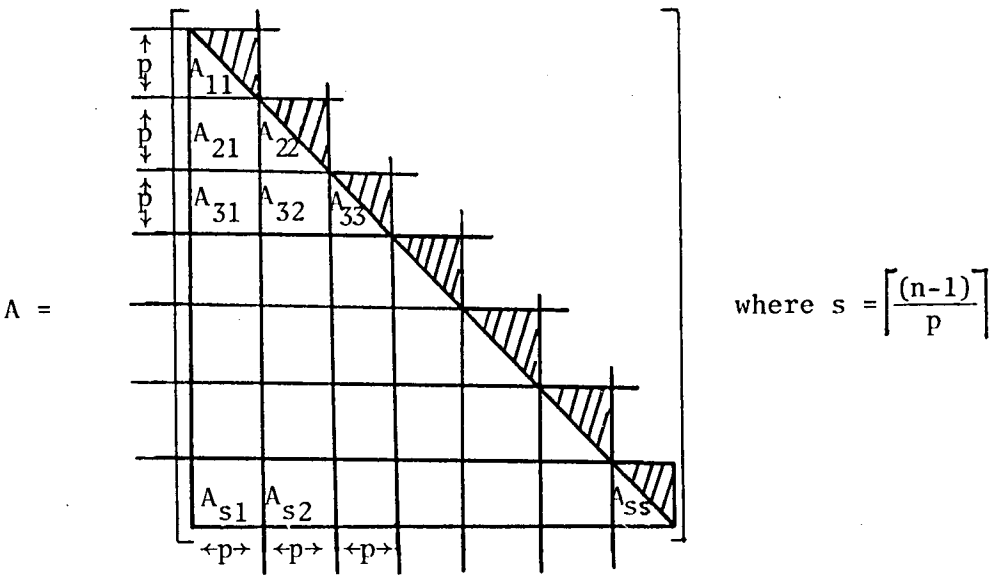
4.3 METHODS THAT REQUIRE AT MOST (n-1)/2 PROCESSORS

The following methods are characterised by the way in which they partition the matrix A and are essentially a forward substitution process with a particular order in which the y_i's are substituted into the system.

They also tend to be less efficient when (n-1) is not exactly divisible by p.

Method 1

Let the matrix A be partitioned as follows:



Partitioning of Matrix A

FIGURE 4.2

The matrix is partitioned into (p×p) blocks where p is a factor of (n-1), (row 1 having been ignored since it is zero).

If we commence with the top left hand block A_{11}, we simply solve for each block one at a time using p processors on each block. We may proceed either by columns thus

```
'FØR' J=1 'STEP' 1 'UNTIL' (N-1)/P 'DØ'
'FØR' I=J 'STEP' 1 'UNTIL' (N-1)/P 'DØ' SØLVE(A[I,J])
```

or by rows thus

```
'FØR' I=1 'STEP' 1 'UNTIL' (N-1)/P 'DØ'
'FØR' J=1 'STEP' 1 'UNTIL' I 'DØ' SØLVE(A[I,J])
```

where the subprogram SØLVE may be defined as follows.

When $A_{i,i}$ is a diagonal block we can treat it as a triangular system and, as there are p processors available, we may assign one to each row of the block and use the forward substitution technique.

Off-diagonal blocks $A_{i,j}$ $(i>j)$ are square sub-matrices where the associated y_i's are known so that all that is required is the substitution of these values into the equations. Again, with p processors available, we may assign one to each row of the block.

The shaded areas in Figure 4.2 represent the places at which processors lie idle and it can be seen that they all appear along the diagonal.

It is clear that each block can be solved in 2p unit steps and since there are $\frac{1}{2}\frac{(n-1)}{p}\left[\frac{(n-1)}{p}+1\right]$ blocks then,

$$T_p = \frac{1}{2}\frac{(n-1)}{p}\left[\frac{(n-1)}{p}+1\right] 2p$$

$$= \left[\frac{(n-1)^2}{p} + (n-1)\right] \quad \text{unit steps,} \qquad (4.3.1)$$

giving

$$S_p = \frac{np}{(n+p-1)} \quad . \qquad (4.3.2)$$

Also,

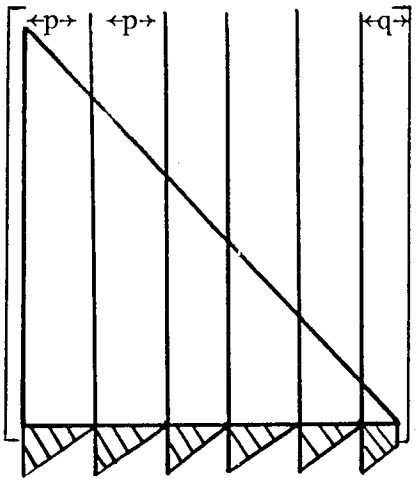
$$E_p = \frac{n}{(n+p-1)} \quad , \qquad (4.3.3)$$

and

$$PF_p = \frac{n^2 p}{(n+p-1)^2} \quad . \qquad (4.3.4)$$

In the event that p is not a factor of (n-1), the final row of blocks will have less than p rows and so create a small additional inefficiency. It is better for this final row of blocks to be as full as possible.

This difficulty may be overcome by slightly altering our approach. If A is partitioned into columns of width p thus:-

A = 

where q<p

Partitioning of Matrix A

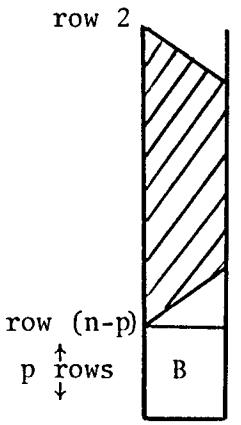
FIGURE 4.3

Then, commencing with the left most column, the p processors are assigned to rows 2 to (p+1). Row 2 is completed first so that processor 1 may be re-assigned to row (p+2). Processor 2 is the next one that becomes available and is re-assigned to row (p+3) and so on until row n is reached. As the processors are available, they await the completion of the column and then move to the next column. The areas of inefficiency now appear at the bottom of each column but the additional inefficiency now only occurs in the final column where there are less than p rows. The T_p,S_p,E_p and PF_p will be as defined in (4.3.1) to (4.3.4). A similar result may be obtained by applying the same process without partitioning A. The area of inefficiency will then appear when substituting values into the final rows of A.

Method 2

This method has a more complicated strategy and is only suitable for odd values of p. Once again A is partitioned into columns of width p and commencing with the left most column, the processors are assigned to the first p rows. We then proceed, as in the second

strategy of method 1, by reassigning processors as they become available until row (n-p) is reached.



(a) A Column of Width p of Matrix A

$$B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & & \\ \vdots & & & \vdots \\ b_{p1} & & \cdots & b_{pp} \end{bmatrix}$$

where

$b_{i,j} = a_{\ell,k}$

$\ell = n-p+i$

$k = cp+j$

(b) Submatrix B

FIGURE 4.4

We have reached the point shown in Figure 4.4(a) where the shaded area represents the substitutions that have already been made. As processors now become available they pass down the columns of submatrix B substituting in values of y, starting with the first column. As the p^{th} processor is released the processors will have come into line and they may then sweep across the remainder of B, thus completing the column without any of the processors becoming idle.

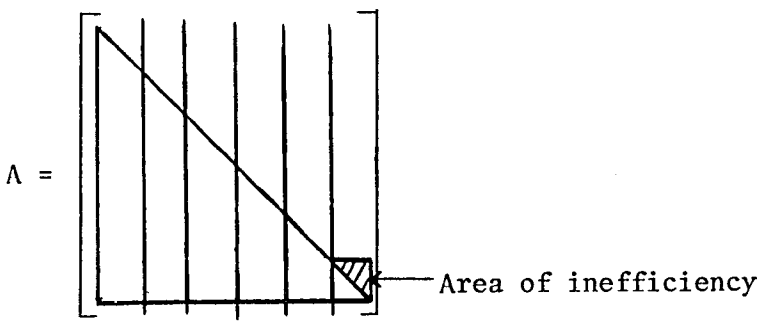
Considering submatrix B, for one algorithm step we have one processor assigned to the submatrix, for the next algorithm step we have two processors and so on until there are (p-1) processors assigned to the submatrix B. In that period a total of t_B where

$$t_B = \sum_{i=1}^{p-1} i = \frac{p(p-1)}{2} \qquad (4.3.5)$$

substitutions will have been made into the submatrix B. Now, since there are p elements per column of B, we must have substituted the y's into (p-1)/2 columns. Thus, if p is odd then an integral number of columns of B will have been completed. Hence as the p^{th} processor completes the substituting of values into row (n-p), there will be

(p+1)/2 complete columns remaining in submatrix B. The p processors are then assigned, one to a row, and sweep across the remainder of B substituting in values of y.

The columns are clearly completed without any processors lying idle except of course the final column which is solved as a triangular system with sufficient processors to assign one to each row. Again in Figure 4.5 we can observe that the inefficiency indicated by the shaded area, is clearly minimal when p is small.



The Partitioning of A

FIGURE 4.5

It is obvious that the final column requires 2p unit steps to be solved and since there will be no redundancy during the processing of the other columns we can say that

$$T_p = [n(n-1)-p(p+1)]/p + 2p$$

$$= [\frac{n(n-1)}{p} + p - 1] \quad \text{unit steps,} \tag{4.3.6}$$

giving

$$S_p = \frac{p.n.(n-1)}{[n(n-1)+p(p-1)]} \quad , \tag{4.3.7}$$

$$E_p = \frac{n(n-1)}{[n(n-1)+p(p-1)]} \tag{4.3.8}$$

and

$$PF_p = \frac{p.n^2(n-1)^2}{[n(n-1)+p(p-1)]^2} \quad . \tag{4.3.9}$$

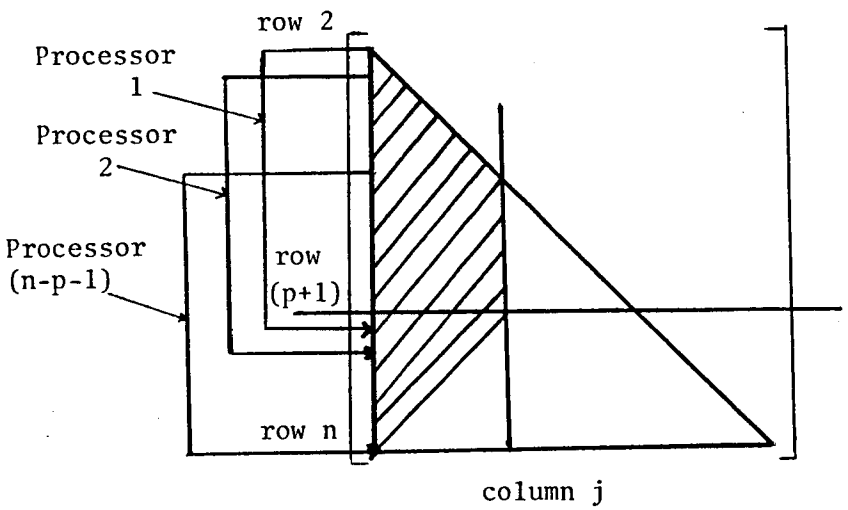
It is clear that these are merely a few of the strategies that may be used for this particular range of p, but because of the minimal area of inefficiency not many will be an improvement on method 2. When considering efficiency these methods are best suited for small values of p. When p is small there is little difference between the performances of methods 1 and 2 so, as method 1 is simpler, it would be recommended.

4.4 THE WAVEFRONT METHODS

In this section, two methods are introduced that require p processors where p lies in the range $(n-1)/2<p<(n-1)$. The methods are the Parallel Wavefront and Delayed Wavefront Methods.

Method 3 - The Parallel Wavefront Method

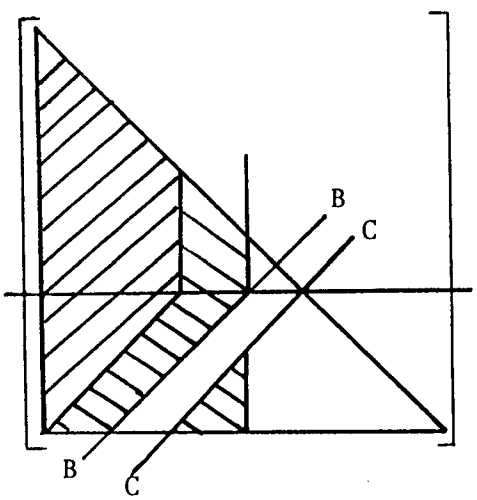
This method is comprised of three distinct phases. During the first phase, as in the previous methods, the p processors are assigned to rows 2 to p+1. After y_1 has been substituted into row 2, processor one becomes available and is reassigned to row p+2 and so on, until we reach the position shown in Figure 4.6. The j^{th} processor (j=n-p-1) has



The Parallel Wavefront Method (end of phase 1)

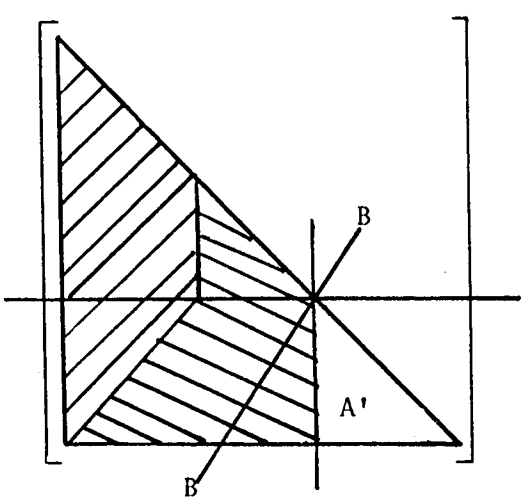
FIGURE 4.6

been reassigned to row n and processors (j+1) to p have reached column j. This position represents the completion of phase 1 and the start of phase 2. Now when processor (j+1) becomes available it is reassigned to row n at column (j+1) and next, processor (j+2) is reassigned to row n-1 at column (j+2) and so on. These new inner products are stored in an auxiliary vector. As this phase progresses we have the situation as seen in Figure 4.7. The 'wavefront' BB is approaching 'wavefront' CC and eventually wavefront BB reaches CC as seen in Figure 4.8.



Phase 2 of the Parallel Wavefront Method

FIGURE 4.7



Parallel Wavefront Method (end of phase 2)

FIGURE 4.8

At this point the y_i (i=p+2 to n) will have been accumulated in two parts and these are now added together.

Now, the third and final phase of the method is commenced. The remainder of the system is a triangular submatrix A' of less than p rows and may be solved as a triangular system with sufficient processors to assign one to each row.

Clearly we have

$$T_p = 2(n-1) + 1$$

$$= (2n-1) \quad \text{unit steps} , \tag{4.4.1}$$

$$S_p = \frac{n(n-1)}{(2n-1)} , \tag{4.4.2}$$

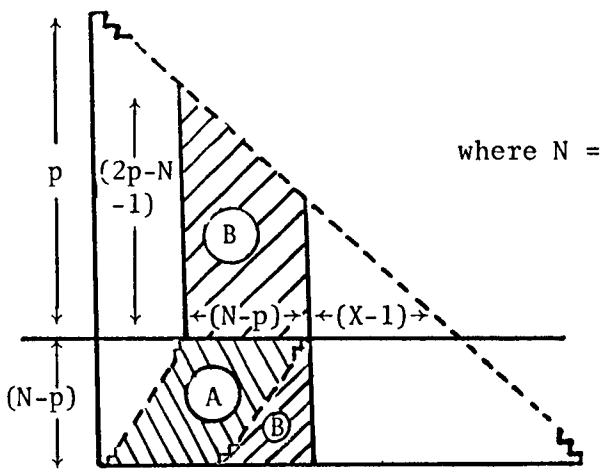
$$E_p = \frac{n(n-1)}{p(2n-1)} \tag{4.4.3}$$

and

$$PF_p = \frac{n^2(n-1)^2}{(2n-1)^2 p} \tag{4.4.4}$$

Since the time T_p is independent of p and from (4.4.3), it is clear that the smaller p is, the more efficient the algorithm becomes, and so, to optimise its performance, the minimum number of processors required by the algorithm must be found, which in turn maximises the efficiency.

Let us assume that on completing phase 2 of the algorithm, X more values have to be substituted into the $(p+2)^{th}$ equation and so consider the following diagram:-



where N = n-1

Matrix A

FIGURE 4.9

Area (A) = $(N-p)^2$ represents the work done by (n-p) processors during the second phase of the algorithm and the duration of the phase $T_p^{(A)}$ is given by:

$$T_p^{(A)} = \frac{(N-p)^2}{(N-p)} = (N-p) \quad . \tag{4.4.5}$$

Combined Areas (B) represent the work done by the remaining (2p-N) processors during the same period, the duration of which is,

$$T_p^{(B)} = [(N-p)(N-p+1)+(2p-N-1)(2p-N)-X(X-1)]/2(2p-N) \quad , \tag{4.4.6}$$

where X is as in Figure 4.9.

Since these times are equal we have,

$$T_p^{(A)} = T_p^{(B)} \quad , \tag{4.4.7}$$

from which we have,

$$2(N-p)(2p-N) = (N-p)(N-p+1)+(2p-N-1)(2p-N)-X(X-1)$$

which reduces to

$$[X-(2N-3p+1)][X+(2N-3p)] = 0 \tag{4.4.8}$$

i.e.,
$$X = (3p-2N) \text{ or } (2N-3p+1) \tag{4.4.9}$$

It has been assumed that $X \geq 0$ but consider the case when $X<0$. This implies that y_1 to y_{p+1} are evaluated before the completion of phase two of the algorithm. At the $(p+1)^{th}$ step of the algorithm y_{p+1} is substituted into rows 2(n-p) to n of A but not into the equation for y_{p+2} until step (p+2). However, at step (p+2), y_{p+2} should be available for substitution into rows 2(n-p)-1 to n and so $X \neq 0$, i.e.,

$$X \geq 0 \quad . \tag{4.4.10}$$

Combining this result with equation (4.4.9) we have either:

$$(3p-2N) \geq 0$$

$$\Rightarrow \quad p \geq \frac{2}{3} N \tag{4.4.11}$$

or

$$(2N-3p+1) \geq 0$$

$$\Rightarrow \quad p \leq \frac{1}{3}(2N+1) \quad . \tag{4.4.12}$$

Obviously the condition (4.4.12) is meaningless and so the condition (4.4.11) is the required one. This gives the following minimum value of p

$$\min (p) = \frac{2}{3} N$$
$$= \frac{2}{3} (n-1) \quad . \tag{4.4.13}$$

This condition may be verified by the following sequences of diagrams. We shall consider a (10×10) system of the form (4.2.2) and indicate the progress of the processors over the array A. The processors are numbered 1,2,...p, their position indicating the elements of A currently being used in the substitution process, and the *'s indicate those elements already used. The y_i whose evaluation is currently being completed is arrowed.

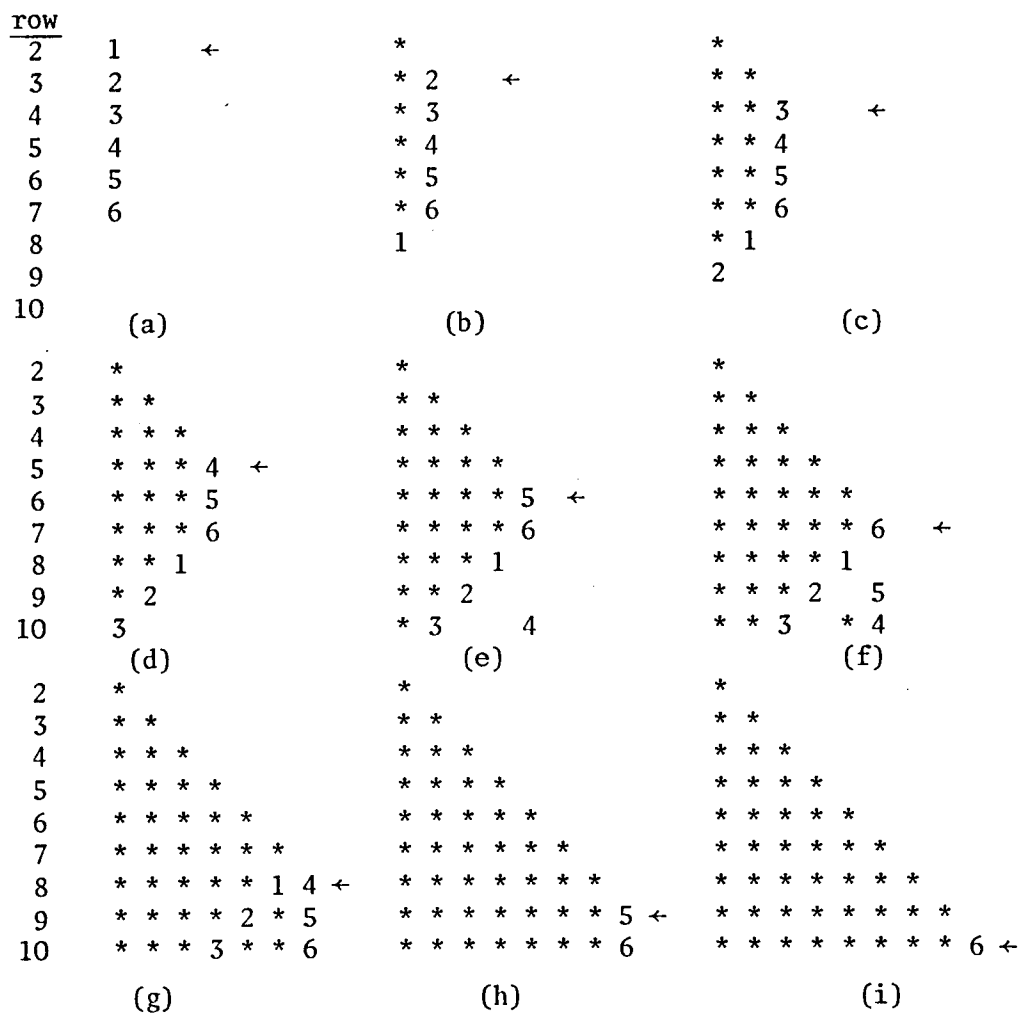


FIGURE 4.10

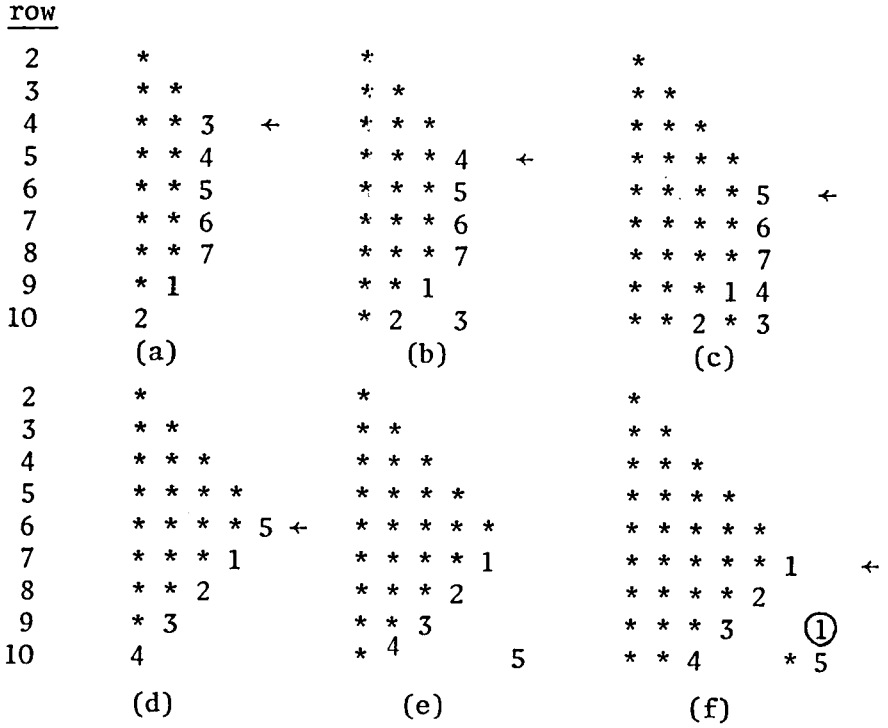


FIGURE 4.11

In Figure 4.10 the sequence of diagrams (a) to (i) represent the solution of the system of equations with the minimum number of processors which is 6. Diagram (d) represents the stage shown in Figure 4.6 and diagram (g) indicates the same stage as in Figure 4.8. It is at this point that the two 'wavefronts' meet and the two parts of the vector elements y_i (i=p+2(1)n) are added together. In the accompanying diagrams 4.10 (h) and (i), the processors sweep across the remaining rows and columns.

Diagrams (a), (b) and (c) of Figure 4.11 illustrate the case when $p>\frac{2}{3}(n-1)$, in this example, p=7 and diagrams (d), (e) and (f) illustrate the case when $p<\frac{2}{3}(n-1)$, here p=5. In the latter example we proceed to diagram 4.11(d) which is the same stage as in diagram (d) of Figure 4.10 and then to diagram 4.11(e). The next stage is seen in diagram (f) of Figure 4.11; we observe that, as expected, an attempt is made to re-assign processor 1 to column 7 of row 9 but it is still occupied with row 7. Arising from this it is clear that y_7 is still being calculated

while an attempt is being made to substitute y_7 into rows 9 and 10, thus an incorrect value of y_7 will be used.

In a simulation of this method when $p \geq \frac{2}{3}(n-1)$ the correct results were produced but when $p < \frac{2}{3}(n-1)$ the values of y_i for i=p+2(1)n were found to be incorrect. This agrees with the assumptions previously stated.

Method 4 - The Delayed Wavefront Method

The delayed wavefront method is the parallel wavefront method adapted for the case when $p < \frac{2}{3}(n-1)$. If we reconstruct the parallel wavefront problem, then during the second phase of operations we reach the point indicated in Figure 4.12.

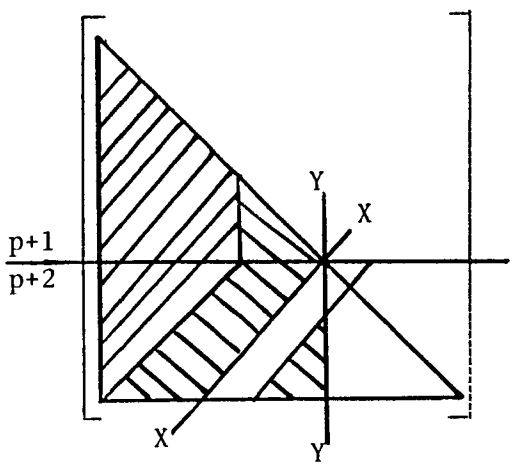


FIGURE 4.12

At this stage the components y_i for i=1(1)p+1 have been evaluated, and in the next algorithm step y_{p+1} is substituted in the equations along wavefront YY while y_p is substituted into the equation for y_{p+2}.

With the parallel wavefront method, y_{p+2} is substituted into the equations along YY during the following algorithmic step. At the same time however, y_{p+1} is being substituted into the equation for y_{p+2}, y_p into the equation for y_{p+3} etc. So y_{p+2} is not available until the following algorithmic step and thus the substitutions along YY must be

delayed for one algorithmic step until y_{p+2} has been calculated. Likewise, with the substitution of y_{p+3} along wavefront YY, the delaying process has to be repeated. The delaying procedure has to be continued until the stage represented in Figure 4.13 is reached.

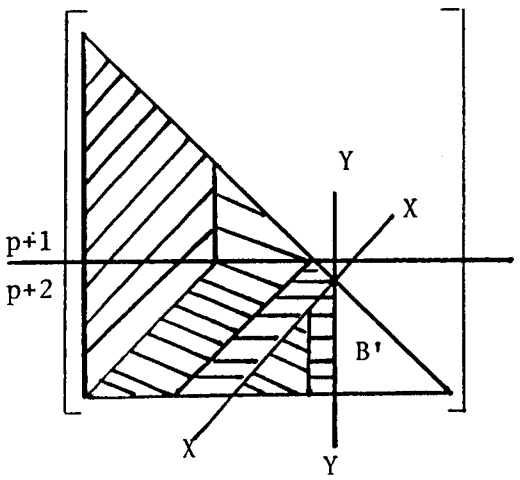


FIGURE 4.13

The equivalent position of the parallel wavefront method is shown in Figure 4.8. At this stage the y_i (i=1(1)n-m) will have been evaluated, where:

$$m = \begin{cases} (p-1)/2 & \text{for p is odd} \\ (p-2)/2 & \text{for p is even} \end{cases} \qquad (4.4.14)$$

The remaining y_i components (i=n-m+1 to n) will have been accumulated in two parts which are now added together. The algorithm is then completed, as in the parallel wavefront method, by treating the remainder of the system of equations (B') as a triangular system with sufficient processors to assign one to each row.

On the completion of the first phase of the algorithm (Figure 4.6) (n-p) algorithmic steps will have been performed. As the second phase begins, the wavefronts will be (n-p-1) algorithmic steps apart. After

a further (2p-n) algorithmic steps are performed the stage represented in Figure 4.12 is reached and the wavefronts will now be (2n-3p-1) algorithmic steps apart. Obviously to reach the stage represented in Figure 4.13 requires a further (2n-3p-1) algorithmic steps. During this delayed phase, wavefront YY advances (2n-3p-1)/2 algorithmic steps when p is odd and (2n-3p)/2 steps when p is even. One unit step is required to 'add the wavefronts' together and the remainder of the algorithm requires (p-1)/2 algorithmic steps when p is odd and (p-2)/2 steps when n is even.

Thus, the time required by the algorithm in unit steps is:

$$T_p = \begin{cases} 2[(n-p)+(2p-n)+(2n-3p-1)]+1+(p-1) & \text{for p is odd} \\ 2[(n-p)+(2p-n)+(2n-3p-1)]+1+(p-2) & \text{for p is even} \end{cases} \qquad (4.4.15)$$

i.e.,

$$T_p = 4n-3p-\ell \quad \text{unit steps} \qquad (4.4.16)$$

$$\ell = \begin{cases} 2 & \text{for p is odd} \\ 3 & \text{for p is even.} \end{cases}$$

Clearly if $p \geq \frac{2}{3}(n-1)$ then the algorithm becomes the parallel wavefront method, but what happens when $p < \frac{(n-1)}{2}$?

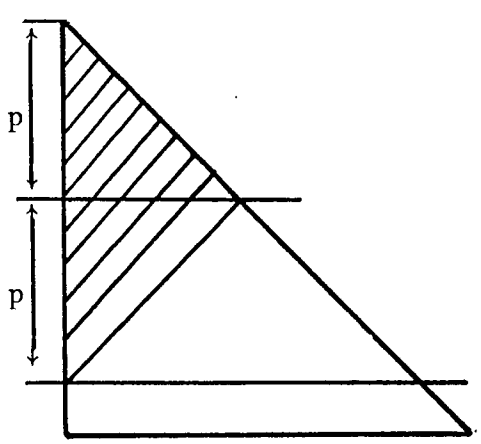


FIGURE 4.14

During the first phase, the stage shown in Figure 4.14 is reached. It will be another algorithmic step before the next processor is released but one is required immediately and so the algorithm breaks down because of insufficient processors being available.

In Figure 4.15, the diagrams (a) to (d) illustrate the delayed wavefront algorithm successfully solving a 10×10 system with p=5. Diagram (a) is identical to Figure 4.11(e) but instead of the situation of Figure 4.11(f) arising, the delaying technique is applied during the next step as in diagram (b) of Figure 4.15. This occurs again in diagram (d). At this point the two waveformts are 'added together' and the algorithm passes on to its final stage successfully.

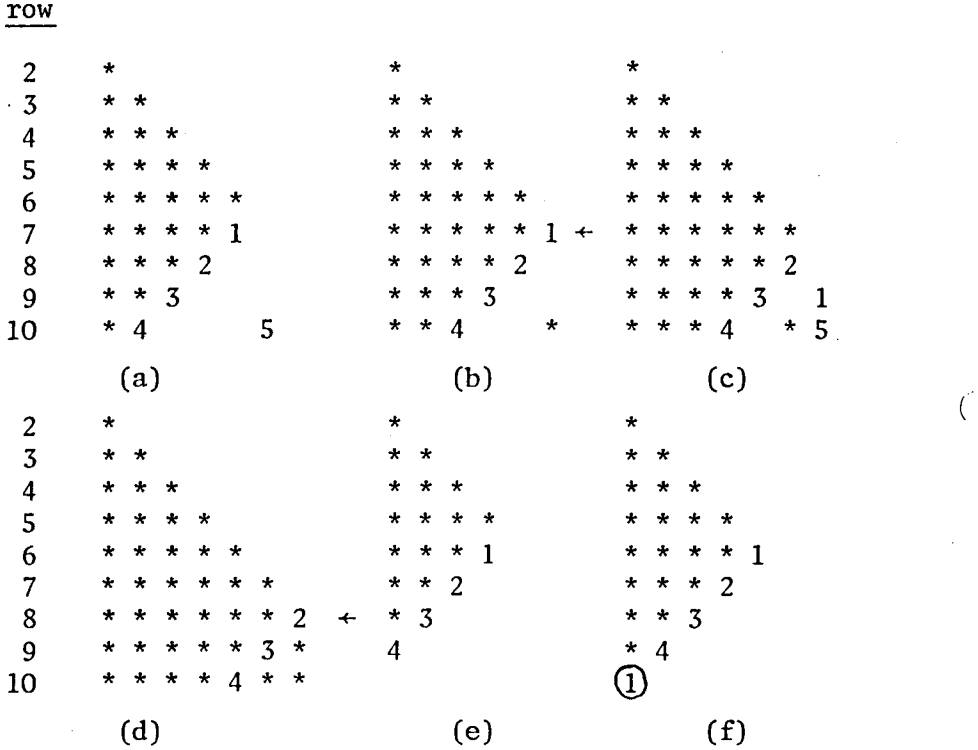


FIGURE 4.15

Diagrams (e) and (f) of Figure 4.15 represent the case when $p<(n-1)/2$, in this instance p=4. The stage shown in Figure 4.14 is illustrated in diagram (e), and it can clearly be seen in diagram (f)

that at the next step an attempt is made to reassign processor 1 to row 10 when it is already assigned to row 6.

4.5 METHODS EMPLOYING MORE THAN (n-1) PROCESSORS

It has already been stated that the maximum number of processors that may be efficiently employed in executing the forward substitution process is (n-1) and that the algorithm requires a minimum of 2(n-1) unit steps. Thus, to solve the problem (4.2.2) in fewer steps and employ more than (n-1) processors efficiently would require restructuring of the algorithm completely. This has been achieved by Chen and Kuck [1975], Heller [1974b], Borodin and Munro [1975] and Orcutt [1974]. The particular algorithm that will now be considered is that of Chen and Kuck since its performance is as good as, if not better than, the other algorithms mentioned.

Method 5

As with most algorithms of this kind, the algorithm of Chen and Kuck requires $O(n^3)$ processors but reduces the unit steps to $O(\log_2^2 n)$. This is achieved at the expense of increasing the total amount of work done in the execution of the algorithm but at the same time increasing the 'amount of parallelism'.

The algorithm of Chen and Kuck can be described as follows:

1. Let B be a lower triangular matrix of order $(n \times n)$ in which the j^{th} column contains $a_{i,j-1}$ for $j \leq i \leq n$, where $a_{i,0}=d_i$.

i.e.

$$B=\begin{bmatrix} d_1 & & & & \\ d_2 & a_{21} & & & \\ d_3 & a_{31} & a_{32} & & \\ \vdots & \vdots & & \ddots & \\ d_n & a_{n,1} & \cdots & & a_{n,n-1} \end{bmatrix} . \qquad (4.5.1)$$

2. Let C be an alias for B, i.e. B and C represent the same memory locations

$$B \equiv C \quad . \tag{4.5.2}$$

3. Repeat this step for $i=1,2,\ldots,\log_2 n$:

   a) Set $k=2^i$;

   b) Partition B and C as shown in Figure 4.16;

   c) Compute $S_j=S_j+T_j*Q_j$, for $i \leq j \leq n/k$;

   simultaneously.

4. The first column of B contains the solutions x_i for $1 \leq i \leq n$.

The number of unit steps is found as follows:
At each iteration during step 3, there is one multiplication followed by the summation of (k/2+1) operands which may be done in at most $\log_2(k/2+1)$ unit steps which is less than or equal to $\log_2 k$ unit steps. Since step 3 is repeated $\log_2 n$ times, we have

$$T_p \leq \sum_{j=1}^{\log_2 n} j + \sum_{j=1}^{\log_2 n} 1 = \frac{1}{2}\log_2^2 n + \frac{3}{2}\log_2 n \tag{4.5.3}$$

which gives,

$$S_p = \frac{2.n(n-1)}{(1\log_2^2 n+3\log_2 n)} = O(\frac{n^2}{\log_2^2 n}) \tag{4.5.4}$$

and

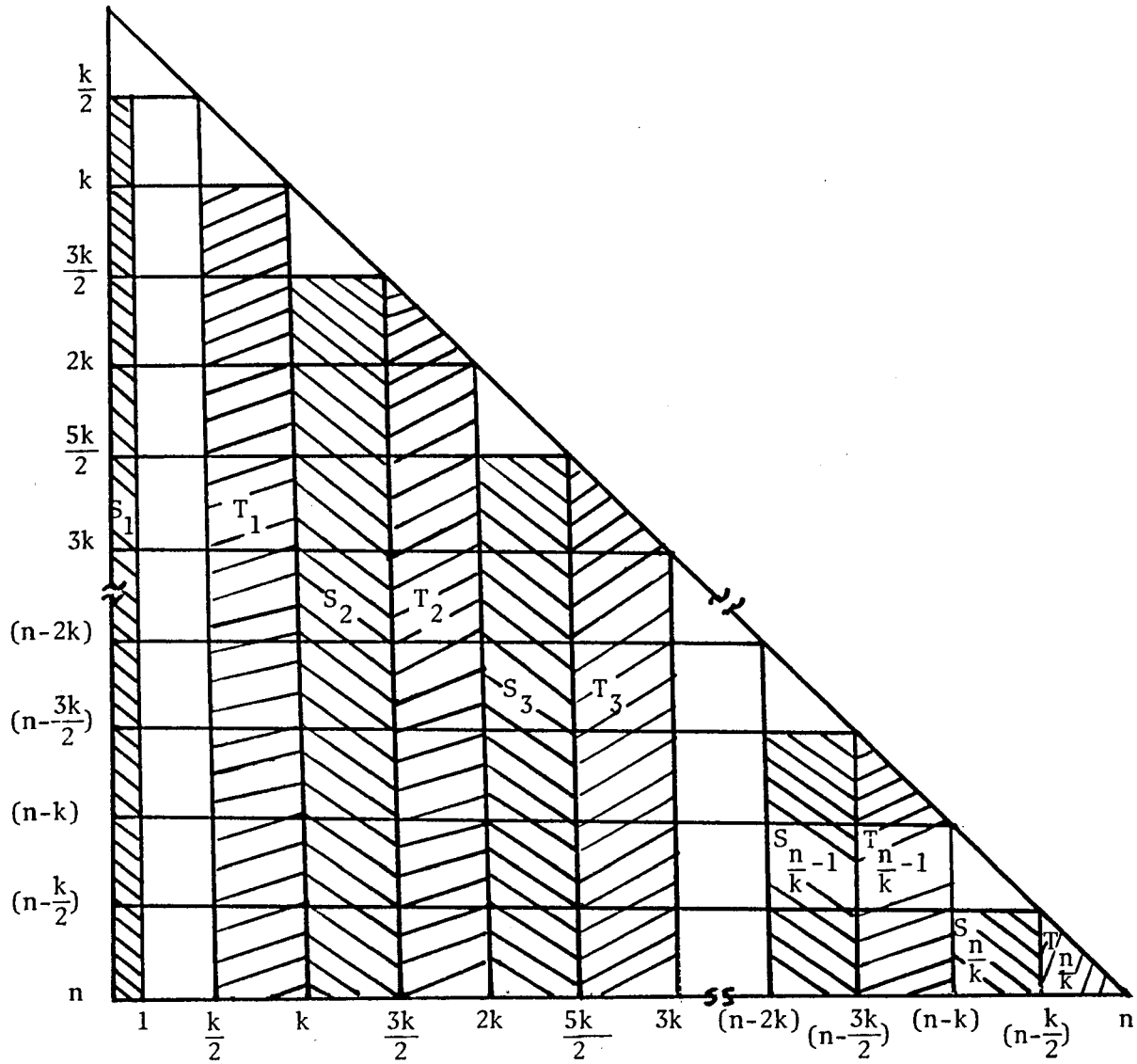
$$E_p = \frac{2.n(n-1)}{P(1\log_2^2 n+3\log_2 n)} \quad . \tag{4.5.5}$$

To calculate the number of processors that are required we refer to Figure 4.16. The maximum number of processors will be needed at the multiplication step of stage 3c, so we simply count the number of multiplications for T_j*Q_j, $1 \leq j \leq n/k$. Thus from Figure 4.16 we have:

$$p(k=2^i) = [\sum_{j=1}^{k/2} j + \frac{k(n-k)}{2}] + \frac{k}{2}[(n/k-1)\sum_{j=1}^{k/2} j + \frac{k}{2}(k+2k+\ldots(n-2k))]$$

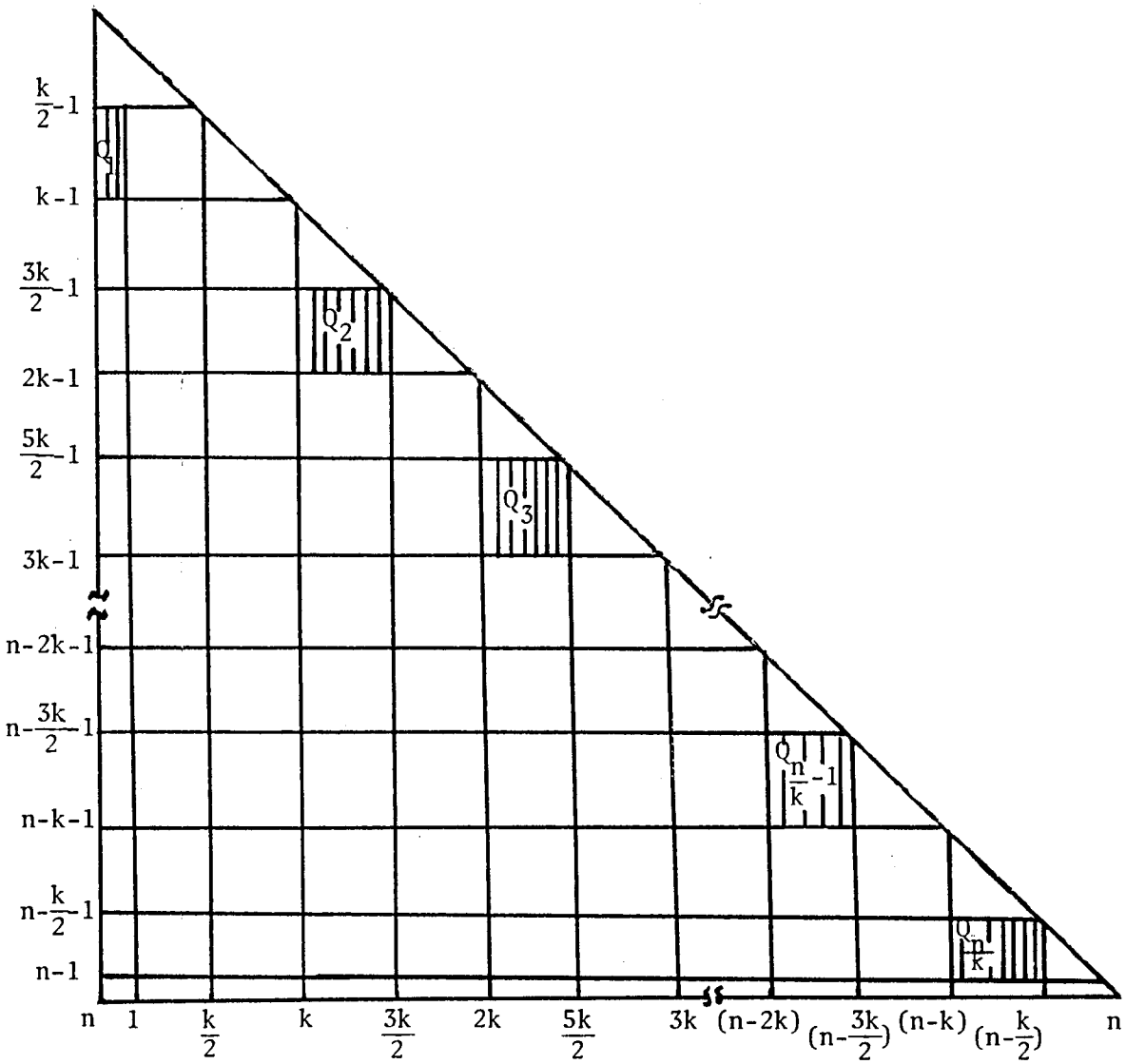
$$\text{for } 1 \leq i \leq \log_2 n$$

$$= \frac{k}{16}[3k^2-(5n+8)k + (2n^2+10n+4)] \quad . \tag{4.5.6}$$



(a) Matrix B

FIGURE 4.16



(b) Matrix C

FIGURE 4.16

When n is large p(k) takes its maximum value during the iteration when k=n/4. This gives the number of processors as,

$$p(n/4) = [\frac{15}{8}n^3 + 16n^2 + 8n]/128 \ . \qquad (4.5.7)$$

Now using these formulae we can produce table 4.1 which shows the performance of the algorithm for different values of n. It is clear that the time T_p and speed-up S_p are very impressive but these results are at the expense of the efficiency which is very disappointing. This also results in very poor performance factors.

n	T_1	p	T_p	S_p	E_p	PF_p
32	992	610	20	49.6	0.08	4.03
64	4032	4356	27	149.33	0.034	5.12
128	16256	32776	35	464.46	0.014	6.58
256	65280	253968	44	1483.64	0.0058	8.67
512	261632	1998880	54	4845.04	0.0024	11.74

TABLE 4.1

The high number of processors that are required certainly makes the algorithm unfeasible on an MIMD type parallel computer and also for existing SIMD type machines. It is doubtful that even future SIMD machines would have sufficient processing elements to cope with the larger systems of equations.

Chen and Kuck [1975] continued by making two suggestions to reduce the number of processors that are required, namely cutting and folding (not to be confused with the folding technique of Chapter 3). First let us consider cutting.

Cutting

The (n×n) matrix B is partitioned or 'cut' into (n/ℓ) columns of width ℓ and each column is processed one at a time. The left most column is comprised of an (ℓ×ℓ) triangular system (at its top) which may be solved by method 5, and an [(n-ℓ)×ℓ] rectangular system R (at

the bottom). The remaining columns are of the same form but with fewer rows in the rectangular system.

The triangular system T may be solved in t_T unit steps as defined in (4.5.3) and will require p_T processors as defined in (4.5.7).

The rectangular systems may be solved by a straightforward substitution of the solution of the corresponding triangular system T into the rows of R and computing the inner products. The rectangular systems will require p_R processors and as the system of the first (or left most) column is the largest we have

$$p_R = (n-\ell)\times\ell \ . \tag{4.5.8}$$

Each system will be solved in the same number of steps t_R where

$$t_R = \lceil \log_2(\ell+1) \rceil + 1 \tag{4.5.9}$$

Obviously the number of processors required by this method will be,

$$p = \max[p_T, p_R] \tag{4.5.10}$$

Also there are (n/ℓ) triangular systems and $(n/\ell-1)$ rectangular systems to be solved so,

$$T_p = (\tfrac{n}{\ell}).t_T + (\tfrac{n}{\ell} -1)t_R \text{ unit steps} \ . \tag{4.5.11}$$

Folding

The second suggestion, folding, is based on the following simple idea. Assume we have a tree of height t which contains (2^t-1) operation nodes and whose evaluation requires 2^{t-1} processors. Obviously the efficiency of such a tree evaluation is,

$$\text{efficiency} = \frac{(2^t-1)}{2^{t-1}.t} \simeq \frac{2}{t} \ . \tag{4.5.12}$$

Now by halving the number of processors, we only require one extra processing step and so,

$$\text{efficiency} = \frac{(2^t-1)}{2^{t-2}(t+1)} \simeq \frac{4}{(t+1)} \ . \tag{4.5.13}$$

When t is large this process approximately doubles the efficiency but has a negligible effect on the speed-up. This process is called a fold.

When i folds are made on a tree of height t, the new tree height is $(t+2^{i+1}-i-2)$ and the processor requirement is reduced to $2^{t-1}/2^{i}$.

Applying this technique to method 5, we have from (4.5.6) that for i folds,

$$p = 2^{i} \sum_{j=1}^{k/2^{i+1}} j\left(1+\frac{k}{2}\left(\frac{n}{k}-1\right)\right) + \frac{1}{2^{i}}\left[\frac{k(n-k)}{2} + \left(\frac{k}{2}\right)^{2}(k+2k+\ldots(n-2k))\right]$$

$$= \left(1+\frac{n}{2}\frac{k}{2}\right) 2^{i} \sum_{j=1}^{k/2^{i+1}} j + \frac{1}{2^{i}}\left[\frac{k}{2}(n-k) + \frac{k^{2}}{2}(k+2k\ldots+(n-2k))\right] . \qquad (4.5.14)$$

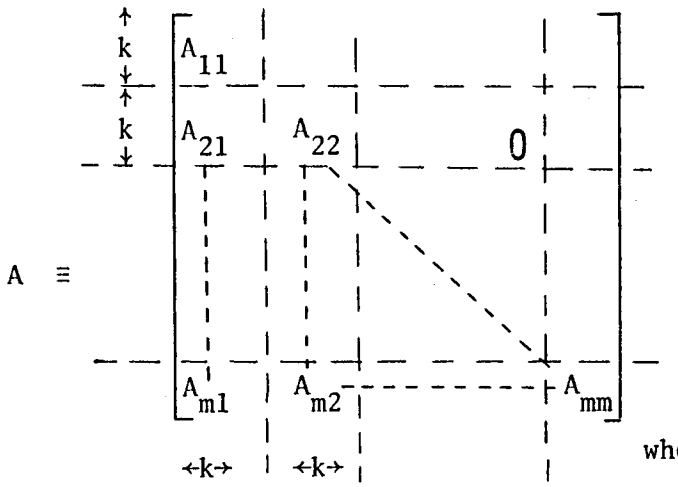
Applying this formula to the step of the algorithm that requires the largest number of processors, we derive a new value for p. Then at every other step in the algorithm for which there are insufficient processors we also apply the folding technique.

It is important to remember that a tree of height t may only be folded (t-1) times and of course a tree of height 1 cannot be folded at all.

It is of course possible to apply a combination of cutting and folding to the algorithm by first applying the cutting technique and then folding as many times as required.

There is one final principle that may be applied to method 5 to improve its performance and that is the problem decomposition principle as suggested by Hyafil and Kung [1974].

The principle is similar to cutting, in that it partitions the matrix A; not into columns however but (k×k) blocks, thus:-



$$A \equiv \qquad (4.5.15)$$

where $n=m\times k$.

Each diagonal block $A_{i,i}$ (i=1(1)m) may be treated as a triangular system and so can be solved by method 5 for which the time $t_{i,i}$ is defined in (4.5.3) and processor requirement $p_{i,i}$ is defined in (4.5.6).

The off-diagonal blocks $A_{i,j}$ (i=2(1)m, j=1(1)i-1) are solved by a straightforward substitution process in a minimum time of

$$t_{i,j} = 1 + \lceil \log_2(k+1) \rceil \quad \text{unit steps,} \qquad (4.5.16)$$

using

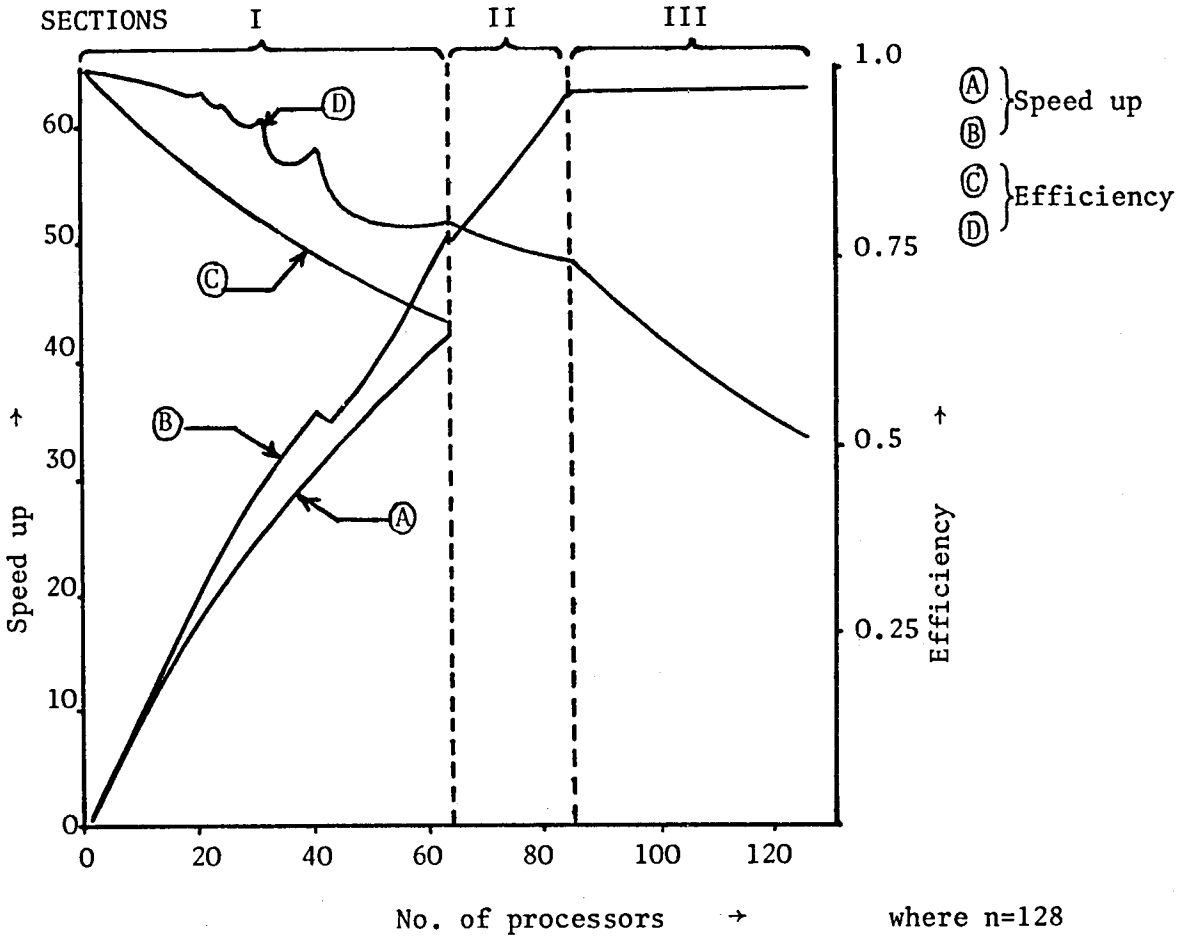
$$p_{i,j} = k^2 \quad \text{processors .} \qquad (4.5.17)$$

Thus, we have that the number of processors required will be $p=\max(p_{i,i}, p_{i,j})$ and the time T_p is

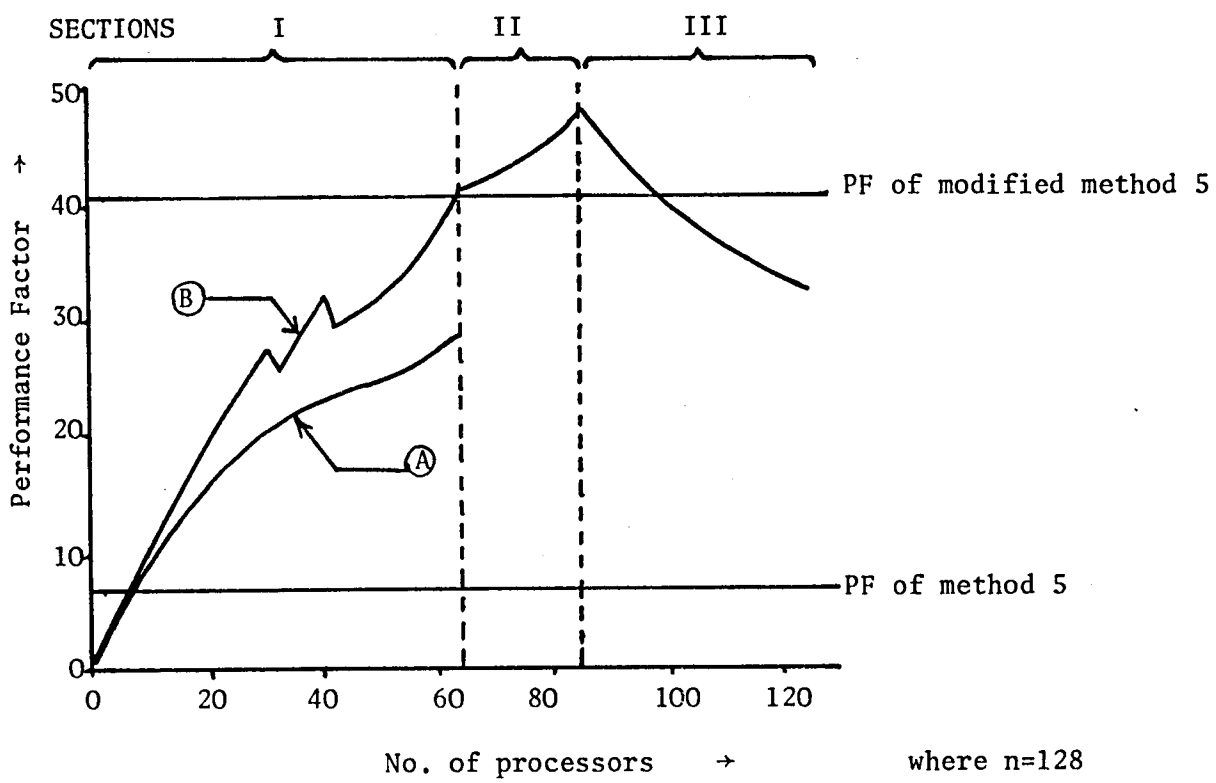
$$T_p = m\, t_{i,i} + \tfrac{1}{2}m(m-1)t_{i,j} \quad \text{unit steps.} \qquad (4.5.18)$$

4.6 RESULTS AND CONCLUSIONS

First of all we shall consider the performance of methods 1,2,3 and 4. The graph 4.1 represents both the speed-up and efficiency of the methods plotted against different values of p for n=128. Graphs (A) and (C) represent the speed-up and efficiency respectively of method 1. Both graphs are smooth because the method is able to cope with the occasions when (n-1) is not exactly divisible by p.



GRAPH 4.1



GRAPH 4.2

Graphs (B) and (D), also representing speed-up and efficiency respectively, may be divided into three sections. Section I from p=1(1)63 represents method 2, section II from p=64(1)84 represents method 4, the delayed wavefront method, and section III from p=85(1)126 represents method 3, the parallel wavefront method.

The uneveness of graphs (B) and (D) in section I is due to the inability of method 2 to cope with the case when (n-1) is not exactly divisible by p entirely satisfactorily but it is still an improvement on method 1.

In section II the curves representing the delayed wavefront method again are smooth as would be expected. Finally in section III the graph (B) becomes a horizontal line since the maximum speed-up for $p<(n-1)$ has been achieved. However graph (D) continues to decrease as p increases since the speed-up is constant while the number of processors increase.

These results are combined in graph 4.2 where the performance factor is plotted against the number of processors. The graphs are divided into sections as in graph 4.1, with (A) representing method 1 and (B) representing methods 2,3 and 4.

Once again we note the uneveness of the section of graph representing method 2 and that for small values of p, the performance factor of method 1 is very close to that of method 2. The graph peaks at p=85 which is the minimum value of p for which the parallel wavefront method may be used. Thus for values of $p<(n-1)$ the parallel wavefront method with $p=\lfloor\frac{2}{3}(n-1)\rfloor$ is the optimum algorithm.

Now let us compare this result with the performance of method 5. The line drawn at PF =6.58 on graph 4.2 represents the performance factor of method 5 which for this order of problem requires 32,776 processors. When the method is modified by cutting and folding etc.,

its performance is improved. There is a second line at $PF=40.32$ which represents the best performance factor of the modified method 5 which in this case requires 256 processors. Clearly, the performance of the parallel wavefront method is still superior to the modified method 5. In fact, the delayed wavefront method also has a better performance than method 5.

A better comparison between method 5 and methods 1,2,3 and 4 can be made by considering table 4.2. In table 4.2, the first column contains the results of method 2 when $p=\lceil\frac{1}{2}(n-1)\rceil$ which from graph 4.2, can be seen is the optimum method for $p<\frac{1}{2}(n-1)$. The second column contains the best results of the delayed wavefront method and the third column the results for the parallel wavefront method with its optimum value of p.

The right-most entry in each row represents the results of method 5 and the remaining entries were made as follows. Results were generated for all the modifications that can be made to method 5. Then, by using the processor count upper limit shown at the head of each column, the results were tabulated for the examples with the best performance factor. Finally, for each row, the example with the best performance factor is indicated.

From the table it can be seen that for problems of order $n\leq256$, the parallel wavefront method has the best performance factor. When $n=512$ however, one entry appears on the right hand side of the table which has a better performance factor than the parallel wavefront method. Although it is an isolated case for $n\leq512$, it can be expected that as the order of the problem increases, similar examples will re-occur.

All the methods presented here satisfy the more stringent requirement of SIMD parallel computers in that only one type of operation is performed at each step.

In Chapter 1, SIMD and MIMD machines were discussed. It was said that existing SIMD computers still have a relatively small number of processing elements i.e., the Illiac IV has 64, but computers under construction have 10,000 processing elements or more. It was emphasised that because of the nature of the computer, speed-up is the essential factor when selecting algorithms. Thus the fastest algorithm should always be selected provided the computer has sufficient processors. So, from table 4.2, when $p \geq n$, method 5 and its modifications are seen to be the best algorithms, although it must be emphasised that the basic method 5 is generally unfeasible because of the large number of processors that are required.

With MIMD machines we are also interested in the efficient use of processors and so the performance factor is more important. Obviously we select the method with the best performance factor which, for problems of order $n<512$, is invariably the parallel wavefront method. When $n=512$ however, we see that the modified method 5 has the best performance factor. It has already been said that this is unlikely to be an isolated case and, for problems of order $n>512$, the modified method 5 is again expected to have the best performance factor. It is also expected that these cases will require a minimum of 2228 processors as is required when $n=512$.

Now MIMD computers tend to have a smaller number of processors than SIMD computers and even the most optimisitic plans for future MIMD computers do not cater for such large processor requirements. So, for MIMD computers, the parallel wavefront method has the best performance factor despite the performance of the modified method 5 for larger triangular linear systems since it requires too many processors.

Another desirable feature of the parallel and delayed wavefront methods are that they automatically adapt to any size of problem. This

means that it does not require the order of the problem to be a power of 2 or 10 or for (n-1) to be exactly divisible by p. Method 5 requires n to be a power of 2 but it is not clear what happens when this is not true.

One final observation is that method 1 has a performance factor that is almost as good as method 2 when p is small compared to n. Then in these cases method 1 would be preferred because of its simplicity. An error analysis has not been included in this chapter but it is planned as future work.

n	BLOCK STRATEGY	DELAYED WAVEFRONT	WAVEFRONT	4	8	16	32	64	128	256	512	1024	2048
32	9.76 12.1 0.81 15	11.56 15.26 0.76 20	11.8 15.75 0.75 21	1.6 2.53 0.63 4		7.59 11.02 0.69 16	7.83 11.53 0.68 17	8.72 23.62 0.37 64	7.33 26.11 0.28 93	5.69 38.15 0.15 256	5.85 43.13 0.14 318	4.03 49.6 0.08 610	
64	19.98 24.89 0.8 31	23.11 30.78 0.75 41	24.00 31.75 0.76 42	1.69 2.6 0.65 4		3.24 7.2 0.45 16	13.5 20.78 0.65 32	18.56 34.46 0.54 64	16.4 45.82 0.36 128	16 64 0.25 256	15.73 70.74 0.22 318	11.57 84 0.14 610	8.73 100.8 0.09 1164
128	40.46 50.48 0.8 63	47.63 63.25 0.75 84	47.81 63.75 0.75 85	1.73 2.63 0.66 4		3.59 7.58 0.47 16		31.51 44.91 0.7 64	35.55 67.45 0.53 128	40.32 101.6 0.4 256	37.94 109.84 0.35 318	32.76 141.36 0.23 610	26.82 176.7 0.15 1164
256	81.41 101.68 0.8 127	95.07 126.76 0.75 169	96 127.75 0.75 170	1.75 2.65 0.66 4		3.79 7.79 0.49 16		8.69 23.58 0.37 64	61.13 88.46 0.69 128	82.57 145.39 0.57 256	81.28 204 0.4 512	79.73 220.54 0.36 610	76.33 298.08 0.26 1164
512	163.97 204.88 0.8 255	191.63 255.25 0.75 340	191.81 255.75 0.75 341	1.76 2.66 0.66 4		3.89 7.89 0.49 16		9.42 24.55 0.38 64		134.49 185.55 0.72 256	166.16 291.67 0.57 512	171.68 419.28 0.41 1024	182.28 460.62 0.4 1164

continued.....

TABLE 4.2

4096	8192	16384	32768	65536	131072	262144	524288	1048576	2097152
8.11 134.4 0.06 2228	5.12 149.33 0.03 4356			Each entry represents $\begin{Bmatrix} P^F \\ S_p^p \\ E^p \\ p \end{Bmatrix}$					
24.91 235.59 0.11 2228	15.78 262.19 0.06 4356	13.51 338.67 0.04 8488	11.03 427.79 0.03 16584	6.58 464.46 0.01 32776					
76.62 413.16 0.19 2228	52.12 476.5 0.11 4356	44.68 615.85 0.073 8488	35.57 768 0.046 16584	20.83 826.33 0.025 32776	15.1 1388.94 0.011 127760	8.67 1483.64 0.006 253968			
209.44 683.11 0.31 2228	163.52 843.97 0.19 4356	148.85 1122.88 0.13 8488	121.91 1421.91 0.09 16584	73.12 1548.12 0.05 32776	49.54 2515.69 0.02 127760	31.68 2843.83 0.011 255264	30.23 3904.96 0.008 504352	21.01 4590.04 0.005 1002528	11.74 4845.04 0.002 1998880

TABLE 4.2 (Continued)

CHAPTER 5

THE PARALLEL QUICKSORT ALGORITHM

5.1 INTRODUCTION

In the previous chapters we have considered important numerical problems and how they may be solved using a parallel computer. To demonstrate the versatility of the MIMD type computer we shall now investigate the computer problem of sorting.

The object of a sorting algorithm is to rearrange the set S_n where,

$$S_n = \{a_1, a_2, \ldots, a_n\} \quad , \tag{5.1.1}$$

into some relative order. The elements a_i, (i=1,2,...n) could be a set of numbers that we wish to arrange in ascending or descending order or a list of names that we require in alphabetical order. However, for the purpose of this investigation we shall assume that the a_i, (i=1,2,...n) are positive integers that we wish to arrange in ascending order.

The problem of sorting on a sequential computer has been investigated by Knuth [1973] who describes only a few of the many algorithms that exist. Unfortunately there is no known 'best' sorting algorithm and we may only conclude that one algorithm is better than others for a particular situation. We shall outline some of these algorithms and then investigate the possibility of restructuring them to produce an efficient parallel sorting algorithm.

A general purpose sorting algorithm is produced which is suitable for execution on a parallel computer. The algorithm which is based on Quicksort (see section 5.2) does not require a fixed number of processors but may theoretically use as many processors as are available. The analysis of the algorithm reveals that there is a maximum number of processors that can be used for a particular size of set S_n and by use of the Performance Factor defined in Chapter 4 we can also demonstrate that there is an optimum number of processors that may be used.

5.2 SEQUENTIAL SORTING ALGORITHMS

We shall now outline some of the more common sequential sorting algorithms that are currently in use which have been chosen for the inherent parallelism that they possess apart from linear insertion.

Linear Insertion

Linear Insertion is the simplest yet one of the most important of sorting techniques and may be described as follows:- assume that the first (i-1) elements of S_n have been sorted, then element a_i may be inserted into its correct place among these elements by comparing it successively with elements $a_{i-1}, a_{i-2}, \ldots$, until an element is found that is less than or equal to a_i, say a_j. The elements $a_{j+1}, \ldots a_{i-1}$ are shifted 'up' one place and a_i is inserted into the $(j+1)^{th}$ position.

It is not difficult to see that the average number of comparisons required to insert element a_i is i/2, so that in order to sort n elements we require on average

$$\frac{1}{2}\sum_{i=1}^{n} i = \frac{n(n+1)}{4} \approx \frac{1}{4}n^2 \quad \text{comparisons} . \qquad (5.2.1)$$

It is obvious that this is also the number of elements we would have to move.

Since the amount of work on average in linear insertion is proprotional to n^2, then it is clear that it is unsuitable for large values of n. However, because it is extremely easy to implement on a computer, it is considered one of the best sorting algorithms for small values of n.

Shell Sort

The Shell Sort or Diminishing Increment Sort (Shell [1959]) is an attempt to improve linear insertion by moving elements more than one position at a time. This is achieved by dividing the set S_n into subsets which are then sorted individually by linear insertion. This process is

then repeated for progressively larger subsets, the final subset being S_n. The subsets are chosen at each stage as follows:-

we select a number d_j $(j=1,2,\ldots\ell)$, where $1=d_\ell<d_{\ell-1}\ldots<d_1\leq n$ and create d_j subsets thus,

$$a_i,\ a_{i+d_j},a_{i+2d_j},\ldots a_{i+n_i d_j} \quad \text{for } i=1,\ldots d_j$$

where $n_i = \left\lfloor \frac{n-i}{d_j} \right\rfloor$.

As the size of the subsets increases, their degree of order also increases and so it is possible to apply linear insertion to the larger subsets without sacrificing efficiency. The selection of the d_j is an important factor in the efficiency of the method but there is no conclusive evidence that any particular choice is best. The method is yet to be completely analysed but Knuth [1973] claims that the amount of work involved is proprotional to $O(n^{3/2})$ for a good choice of d_j.

Bubble Sort

Bubble Sort is an example of sorting by exchanging as opposed to sorting by insertion. During the basic process, a_1 is compared with a_2 and, if they are out of order, they are exchanged with each other. This is repeated with a_2 and a_3, a_3 and a_4, etc. and finally with a_{n-1} and a_n. The whole process is repeated until no more exchanges are necessary.

The name Bubble Sort is derived from the fact that elements tend to 'bubble up' to their correct position. Unfortunately, although the fundamental idea is simple, the method compares very badly with other sorting techniques due to the relatively complex program that it involves(Knuth, 1973).

Quicksort

Quicksort (Hoare, 1962) or partition-exchange sort is considered the best general purpose method for sorting on a computer. The basic process of quicksort places some element of S_n, say a_k, into its correct position in such a way that all the elements to the left of a_k are less than a_k and those to its right are greater than a_k. Thus, the original problem has been reduced to two smaller problems, namely, sorting the left subset (containing all elements less than a_k, i.e., $a_1,a_2,\ldots,a_{k-1}$) and the right subset (containing all elements greater than a_k, i.e., $a_{k+1},\ldots,a_n$). The same process may be applied to each subset and repetition of this technique eventually produces subsets containing only one or no elements, at which point the set S_n is sorted.

The process by which a_k is placed in its correct position, called the partitioning process, involves the use of two pointers, i and j. Initially setting i=1 and j=n, j is repeatedly reduced by 1 until an a_j is found such that $a_j<a_i$. The two elements a_i and a_j are exchanged and i is then repeatedly increased by 1 until an a_i is found such that $a_i>a_j$. The elements a_i and a_j are exchanged and we once again decrease j and so on until i=j(=k). The new element a_k, called the partitioning element, is in fact the original a_1, and it has been moved to its correct position such that $a_i<a_k$ (i=1(1)k-1) and $a_i>a_k$ (i=k+1(1)n).

The overheads involved in the partitioning process make it best suited for large values of n and so, in practice, the process is only applied to subsets above a certain size. Linear Insertion is used to sort the smaller subsets, i.e., the subsets that are of a size such that it is more efficient to sort by Linear Insertion than Quicksort.

These are just a few of the many sorting methods that exist. There are other methods that reduce the number of comparisons and exchanges

to a minimum, but they are so complex that they are impractical to use. The simplicity of Linear Insertion makes it difficult to find a better method for small sets S_n. When n is large, however, Quicksort is generally regarded as the best method.

5.3 SORTING ON A PARALLEL COMPUTER

There is already one parallel sorting method, called Batcher's method (Batcher, 1968) that has been developed. This method is similar to Shell's method but the comparisons are arranged so that they do not overlap and so may be done simultaneously. To achieve a significant speed-up, the method requires O(n/2) processors which means that it is not really suitable for MIMD type computers when n is large.

The obvious way to sort S_n using p processors is to divide S_n into p subsets and sort each subset concurrently using all p processors. The difficulty arises in how best to divide the set S_n up into subsets.

If the subsets are produced by a straightforward division of S_n, then once the subsets have been sorted they must be merged. Any advantage gained during the sorting phase would be lost in the merge phase since it would be difficult to involve p processors in the merging of p subsets.

The inefficient merge phase can be eliminated if the chosen subsets are mutually sorted, i.e., if there are p subsets $SubS_i$ (i=1(1)p) such that,

$$SubS_1 + SubS_2 + \ldots SubS_p = S_n \qquad (5.3.1)$$

then they are mutually sorted if

$$(\text{all elements of } SubS_1) < (\text{all elements of } SubS_2) < \ldots$$
$$\ldots\ldots < (\text{all elements of } SubS_p) \quad . \qquad (5.3.2)$$

Unfortunately it is not easy to produce such subsets. Some sort of selection procedure would be necessary, which apart from

being expensive, would not necessarily produce subsets of equal size since the distribution of S_n is not always known. It is important to have subsets of approximately the same size because if one is considerably larger than the rest, it would dominate the running time of the algorithm. So an initial selection procedure can cause as much harm as the merge phase already mentioned.

It has already been stated that sequential algorithms often conceal their potential parallelism and so we shall examine the sequential sorting algorithms of section 5.2 for inherent parallelism. First consider Linear Insertion which is essentially sequential in nature. It is of course possible to insert more than one element at the same time but this idea can create many additional problems. If, for instance, we attempt to insert two elements into the same position in S_n then one of these elements may be lost while the other is duplicated. To safeguard against such a situation involves a more complicated program and hence makes the method less efficient.

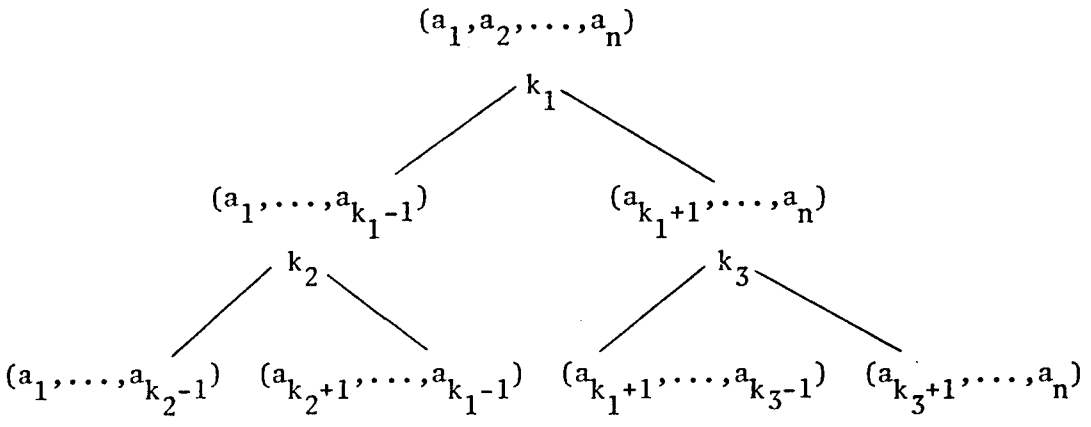
If we consider Shell Sort we see that the sequence d_i (i=1(1)ℓ) produces groups of subsets. Since the subsets in each group are independent (i.e., each element of S_n is a member of one and only one subset), then they may be sorted concurrently. It is important that the subsets produced by d_1 are all sorted before the subsets produced by d_2 are sorted and these, in turn, are sorted before those produced by d_3, and so on. Obviously, a suitable choice of d_i ensures sufficient subsets to occupy all of the p processors. However, in the later stages of the algorithm as d_i decreases (in particular when d_ℓ=1), the number of active processors decreases. Unfortunately the subsets are becoming larger and so are taking longer to be sorted, thus the processors that become idle, will remain idle for a long period.

The next algorithm that we considered was Bubble Sort. As with Linear Insertion the Bubble Sort is essentially sequential since, in the list a_i (i=1,2,...,n), we compare a_2 and a_1 before it is compared with a_3, etc. If we alternately consider the sets of pairs (a_1,a_2), $(a_3,a_4),\ldots,(a_{n-1},a_n)$ and $(a_2,a_3),(a_4,a_5),\ldots,(a_{n-2},a_{n-1})$, then we have formed two sets of independent pairs of elements. This form of the algorithm is similar to Batchers Parallel Sort which is unsuitable for MIMD type computers.

Finally we have the Quicksort algorithm whose partitioning process produces mutually independent subsets which is a very desirable feature. Initially only one processor may be used but, after the first partition has been made, independent subsets are rapdily created. This is the reverse of the case of Shell Sort where it is at the end of the algorithm that the processors become idle. Quicksort has the advantage that, at any stage in the execution of the algorithm, all those subsets that have not been partitioned are independent. This means that, unlike Shell Sort, there is no necessity to sort any subset or group of subsets before others. Thus, on these issues, it was decided to base the parallel sorting method on Quicksort.

5.4 THE PARALLEL QUICKSORT METHOD

The concept of Quicksort is represented diagrammatically in the partition-tree in Figure 5.1. In this figure the first three partitioning stages are shown, where, in the partitioning of the original set S_n, the partitioning element is placed in position k_1, the partitioning element of the left subset is placed in position k_2 and that of the right subset in position k_3.



Partition Tree

FIGURE 5.1

Obviously, the worst running time is achieved when the partitioning procedure produces an empty subset, since it reduces the order of the original problem by only one. In the parallel implementation of Quicksort, there is the added disadvantage that the inherent parallelism of the method is removed, i.e., instead of producing two independent subsets, only one is created. So it is desirable that the choice of partitioning element is as close to the median of the subset being partitioned as possible.

Although the worst running time of the algorithm can never be completely avoided, the possibility of it occurring can be reduced and this is the object of the many variations of the Quicksort algorithm that exist. This objective is achieved by a more careful selection of the partitioning element.

Quicksort and its variations have been thoroughly analysed by Sedgewick [1975] and he concludes that one of the best variations of Quicksort is the median-of-three Quicksort method. The method, originally suggested by Hoare [1962] and later investigated by Singleton [1969], derives its name from the way in which the partitioning

element is selected, being the median of a sample of three elements from the whole subset.

The three elements from which the partitioning element is chosen, are usually the first, middle and last elements of the subset. After they have been mutually sorted the median of the three, the new partitioning element, is exchanged with the second element of the subset. The first and last elements may now be ignored in the partitioning process since we know that they are already in their correct positions in relation to the partitioning element.

A more efficient partitioning process is also adopted, which inserts the partitioning element into its final position at the end of the process rather than being continually moved as previously described. In this process, the pointer i is set to the third element of the subset and pointer j to the next to last element of the subset. Pointer i is increased until an element is found that is greater than the partitioning element and then pointer j is decreased until an element is found that is less than the partitioning element. Obviously, if the subset is to be correctly partitioned, these two elements must be exchanged. The process is continued until the pointers cross, at which point j=i-1. Clearly a_j is the right-most element of the left subset and since the partitioning element is in this subset it is interchanged with element a_j. Thus the partitioning process is completed without unnecessary movement of the partitioning element.

If the situation arises that there are no elements in the subset greater than the partitioning element, then it is possible that the process by which the pointer i is incremented will not be terminated. This may be overcome by creating a dummy element a_{n+1} that is larger than all the other elements. In order to avoid a similar problem with pointer j, another dummy element a_0 is created that is less than

all the other elements.

Let us define two integers ℓ and u such that (ℓ,u) represents a subset containing elements $a_\ell, a_{\ell+1}, \ldots, a_u$. Then the partitioning processes may be described algorithmically as follows:-

Step 1 Sort elements $a_\ell, a_{(\ell+u)/2}, a_u$ into mutual order.

Step 2 Interchange $a_{\ell+1}$ and $a_{(\ell+u)/2}$. Set $i=\ell+1$, $j=u$ and $v=a_i$.

Step 3 Let $i=i+1$. Repeat while $a_i<v$.

Step 4 Let $j=j-1$. Repeat while $a_j>v$.

Step 5 If $i<j$ then interchange a_i and a_j and return to step 3, otherwise proceed to step 6.

Step 6 Interchange $a_{\ell+1}$ and a_j.

This procedure produces the two subsets $(\ell,j-1)$ and $(j+1,u)$.

Clearly, from Figure 5.1, the subset (1,n) is partitioned to produce two subsets $(1,k_1-1)$ and (k_1+1,n) which, in their turn, are partitioned to produce four more subsets. If we have p processors, then we may continue to partition until there are p subsets. These p subsets may then be sorted concurrently using any standard sequential sorting algorithm.

To ensure efficiency, the p subsets must be sorted in approximately the same amount of time which means they must be approximately the same size and have the same degree of disorder. Unfortunately, the partitioning procedure does not guarantee this and so the strategy is not entirely satisfactory.

An alternative strategy is to repeat the partitioning procedure until p subsets are produced, at which point all p processors will be in use. If the process is now continued the number of subsets to be partitioned will be greater than the number of processors, and so the 'extra' subsets are put in a queue until processors become available to partition them.

As with sequential Quicksort, when subsets are small it is more efficient to use Linear Insertion to sort them. The process is terminated when there are no subsets remaining to be sorted by Linear Insertion or partitioning.

Although applying this strategy will not mean that all the processors complete their work simultaneously, it is expected that, since the last subsets to be sorted will be small, the period during which they do complete their work will be a minimal one.

The procedure that executes Parallel Quicksort will have the following basic form:-

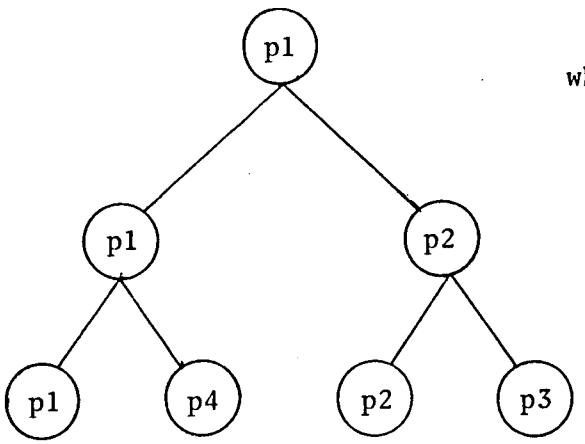
```
'PRØCEDURE' QUICKSØRT(L,U);
'IF' U-L 'GT' M 'THEN';
'BEGIN'
     PARTITIØN(L,U);
     'FØRK' L1,L2;

  L1:QUICKSORT(L,K-1);
     'GØTØ' L3;

  L2:QUICKSØRT(K+1,U);
     'GØTØ' L3;

  L3:'JØIN' L1,L2;
'END'
'ELSE' 'IF' U-L 'GT' 1 'THEN' LINEARINSERT(L,U);
```

where M is the size of the largest subset that is sorted using Linear Insertion, K is the final position of the partitioning element, PARTITION is the partitioning process and LINEARINSERT is a procedure for performing linear insertion.

In standard Quicksort, the smaller of the two subsets produced by the partitioning process is usually sorted first so as to minimise the maximum recursive depth of the algorithm. In Parallel Quicksort this technique minimises the maximum length of the queue of unsorted subsets. It will be seen later that it is preferable to sort the larger of the two subsets first so that the queue is kept as full as

possible. This is to help to avoid periods during which the number of subsets currently being partitioned is less than the number of processors, i.e., to avoid periods when processors become idle.

The Figure 5.2, represents the allocation of processors corresponding to the partition tree of Figure 5.1. In Figure 5.2, it is assumed that the right subset (k_1+1,n) is the smaller subset produced by partitioning S_n. Hence, it is reasonable to expect the partitioning of this subset to be completed before that of the left subset and so processor 2 will request another processor before processor 1 does. Thus processor 3 is assigned to one of the subsets produced by processor 2 and later, when processor 1 requests another processor, it is assigned processor 4, etc..



where pi=processor i.

Allocation of Processors

FIGURE 5.2

5.5 THE ANALYSIS OF THE RUN TIME OF THE PARALLEL QUICKSORT ALGORITHM

In the following analysis of the Parallel Quicksort Method an attempt is made to estimate its run time on a parallel computer with p processors.

The Parallel Quicksort Method consists of three phases that are illustrated in Figure 5.3. Phase 1 or the initial phase is the period

at the beginning of the algorithm when the number of processors in use increases from 1 to p. At first, the increase is gradual, because the subsets being partitioned are large, but later it becomes rapid as the subsets become smaller.

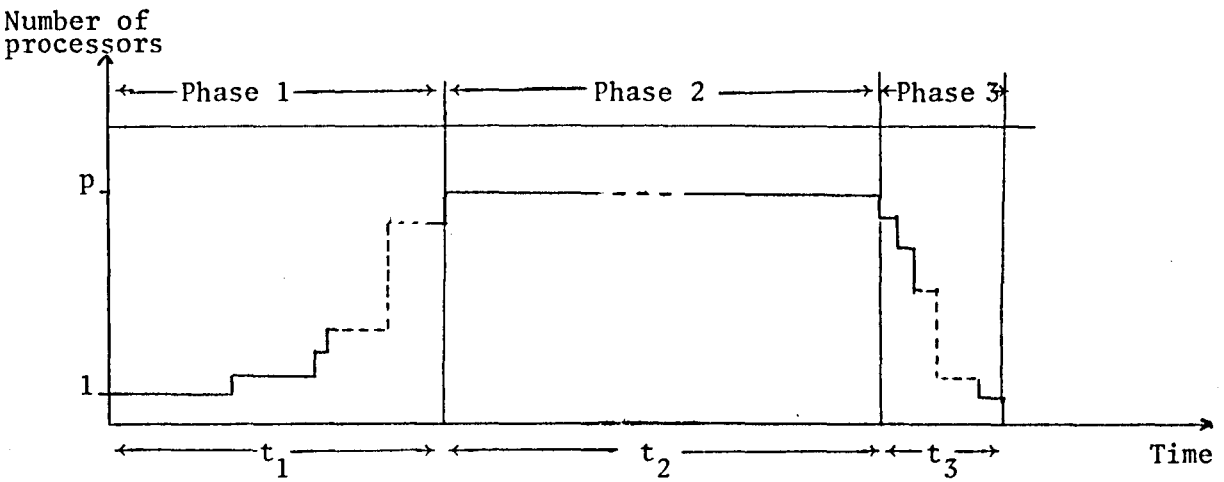


FIGURE 5.3

The second phase or phase 2, is the period during which all p processors are in use and the final phase (phase 3) is the period at the end of the algorithm when the processors become idle. (Although possible, it is not expected that all of the processors will become idle simultaneously).

If the overall run time of the algorithm is T_p and the duration of the i^{th} phase is t_i (for i=1,2,3), then,

$$T_p = t_1 + t_2 + t_3 \quad . \tag{5.5.1}$$

It is not easy to estimate T_p accurately because of the nature of the algorithm. The most difficult time to estimate is t_3, the time between the first processor becoming idle and the p^{th} one becoming idle. Since this phase is relatively short compared to the other two phases it may be ignored.

If the sequential run time of the algorithm and that of phase 1 can be estimated then the run time for phase 2 can be approximated by the following formula:

$$t_2 = \frac{\tilde{t} - \tilde{t}_1}{p} \quad , \tag{5.5.2}$$

where $\tilde{t}$ is the sequential run time of Parallel Quicksort and $\tilde{t}_1$ is the sequential run time of phase 1. The way in which $\tilde{t}_1$ is estimated also yields a simple formula for t_1 but first we must estimate $\tilde{t}$.

The run time of standard Quicksort has been successfully analysed by Sedgewick [1975] by estimating the number of times each statement in the Quicksort program is executed. If a similar technique is applied to the Parallel Quicksort program, it is found that the frequency with which each statement is executed depends on the following quantities:

A - the number of partition stages,

B - the number of exchanges during partitioning,

C - the number of comparisons during partitioning,

D - the number of insertions during linear insertion,

E - the number of elements moved during insertion,

and F - the number of linear insertion stages.

These quantities, except F, are identical to those on which the run time of sequential Quicksort depends.

For the purpose of this analysis we assume that the set S_n contains n distinct elements, with each of the n! permutations of the elements being equally likely. Since the decisions made during the execution of the algorithm are dependent on the elements relative order and not their actual value we further assume that the elements are the numbers (1,2,3,...,n). It is clear also that the subsets produced by partitioning are of a similar form. Also, all of the mathematical results used in this analysis are derived in Appendix A.

To evaluate each of the quantities A,B,C,D,E and F we adopt the same strategy, so let Y represent one of these quantities. Defining Y_n as the average value of Y, then Y_n is obviously equal to the average value of the contribution of the first partitioning stage plus the average value of that quantity required to sort the two subsets. Thus Y_n is defined by the relationship,

$$Y_n = y_n + \sum_{s=1}^{n} \{\text{Probability that s is partition element}\}(Y_{s-1}+Y_{n-s}), \tag{5.5.2a}$$

where s is the partitioning element, Y_{s-1} is the left subset, Y_{n-s} the right subset and y_n the average contribution of the first partition stage to Y_n.

Since s is the median of a sample of three elements, the probability that s is the partitioning element is the proportion of the total number of samples of 3 elements for which s is the median. Clearly, the number of samples for which s is the median is (s-1)(n-s) and the total number of samples of 3 elements is $\binom{n}{3}$, where

$$\binom{n}{k} = \begin{cases} \frac{1}{k!}\prod_{j=1}^{k}(n-k+j) & , \text{ for } k\geq 0 \\ 0 & , \text{ for } k<0 \end{cases} \tag{5.5.3}$$

Thus we have,

$$\{\text{Probability that s is partition element}\} = \frac{(s-1)(n-s)}{\binom{n}{3}} . \tag{5.5.4}$$

Substituting this result into equation (5.5.2a) we have the recurrence relation:

$$Y_n = y_n + \sum_{s=1}^{n} \frac{(s-1)(n-s)}{\binom{n}{3}} (Y_{s-1}+Y_{n-s}), \text{ for } n>m, \tag{5.5.5}$$

where m is the size of the subset above which Quicksort is used; those subsets less than or equal to m in size being sorted by Linear Insertion.

If we consider the sums involving Y_{s-1} and Y_{n-s} separately, it is obvious that they are the same and so equation (5.5.5) may be simplified to,

$$Y_n = y_n + 2 \sum_{s=1}^{n} \frac{(s-1)(n-s)}{\binom{n}{3}} Y_{s-1} , \quad \text{for } n>m. \tag{5.5.6}$$

With a view to solving this equation using generating functions, multiply it through by $\binom{n}{3}z^n$ and sum over all n to give,

$$\sum_{n\geq 0} \binom{n}{3} Y_n z^n = \sum_{n\geq 0} \binom{n}{3} y_n z^n + 2 \sum_{n\geq 0} \sum_{s=1}^{n} (s-1)(n-s) Y_{s-1} z^n , \quad \text{for } n>m.$$

In the second term of the right hand side we can replace s by s+1, n by n+1 and interchange the order of summation so that,

$$\sum_{n\geq 0} \binom{n}{3} Y_n z^n = \sum_{n\geq 0} \binom{n}{3} y_n z^n + 2z \sum_{s\geq 0} \sum_{n\geq s} (n-s)s\, Y_s z^n$$

$$= \sum_{n\geq 0} \binom{n}{3} y_n z^n + 2z\Big(\sum_{s\geq 0} sY_s z^s\Big)\Big(\sum_{n\geq 0} nz^n\Big), \quad \text{for } n>m. \tag{5.5.7}$$

To consider the quantity $z(1-z)^{-2}$, expand it by using Taylor's Theorem to give

$$z(1-z)^{-2} = z(1+2z+3z^2+\ldots)$$

$$= \sum_{n\geq 0} nz^n ,$$

and substituting this result into equation (5.5.7) gives,

$$\sum_{n\geq 0} \binom{n}{3} Y_n z^n = \sum_{n\geq 0} \binom{n}{3} y_n z^n + 2z\Big(\sum_{s\geq 0} sY_s z^s\Big) \frac{z}{(1-z)^2} , \quad \text{for } n>m. \tag{5.5.8}$$

If we define $Y(z) = \sum_{n\geq 0} Y_n z^n$ as the generating function for $\{Y_n\}$, then by differentiating $Y(z)$, with respect to z, j times we have,

$$Y^{(j)}(z) = \sum_{n\geq j} n(n-1)\ldots(n-j+1) Y_n z^{n-j} ,$$

or

$$\frac{Y^{(j)}(z).z^j}{j!} = \sum_{n\geq 0} \binom{n}{j} Y_n z^n .$$

Substituting this result into equation (5.5.8) gives,

$$\frac{Y^{(3)}(z)z^3}{6} = \sum_{n\geq 0} \binom{n}{3} y_n z^n + 2\frac{z^3 Y^{(1)}(z)}{(1-z)^2} \text{ for } n>m,$$

and multiplying through by $(1-z)^3/z^3$ we obtain,

$$\frac{(1-z)^3 Y^{(3)}(z)}{6} = \frac{(1-z)^3}{z^3} \sum_{n\geq 0} \binom{n}{3} y_n z^n + 2Y^{(1)}(z)(1-z), \text{ for } n>m. \tag{5.5.9}$$

The following manipulations may be simplified by changing the variable z to x=(1-z) and defining f(x)=Y(1-x), then,

$$-\frac{x^3 f^{(3)}(x)}{6} = \frac{x^3}{(1-x)^3} \sum_{n\geq 0} \binom{n}{3} y_n (1-x)^n - 2xf^{(1)}(x), \text{ for } n>m,$$

or
$$2xf^{(1)}(x) - \frac{x^3 f^{(3)}(x)}{6} = \frac{x^3}{(1-x)^3} \sum_{n\geq 0} \binom{n}{3} y_n (1-x)^n, \text{ for } n>m. \tag{5.5.10}$$

Introducing the operator θ, defined as

$$\theta f(x) = xf^{(1)}(x) ,$$

then we have,

$$\theta(\theta-1)f(x) = \theta(xf^{(1)}(x)-f(x)) = x^2 f^{(2)}(x)$$

and
$$\theta(\theta-1)(\theta-2)f(x) = \theta(\theta-1)(xf^{(1)}(x)-2f(x)) = x^3 f^{(3)}(x).$$

Substituting these results into equation (5.5.10) gives,

$$2\theta f(x) - \frac{\theta(\theta-1)(\theta-2)}{6} f(x) = \frac{x^3}{(1-x)^3} \sum_{n\geq 0} \binom{n}{3} y_n (1-x)^n, \text{ for } n>m,$$

which leads to

$$(-\theta)(-\theta-2)(5-\theta)f(x) = \frac{6x^3}{(1-x)^3} \sum_{n\geq 0} \binom{n}{3} y_n (1-x)^n, \text{ for } n>m \tag{5.5.11}$$

or in the original variable z,

$$(-\theta)(-\theta-2)(5-\theta)Y(z) = 6\frac{(1-z)^3}{z^3} \sum_{n\geq 0} \binom{n}{3} y_n z^n , \text{ for } n>m . \tag{5.5.12}$$

Now, by definition we have,

$$f(x) = Y(1-x) = \sum_{n\geq 0} Y_n (1-x)^n ,$$

so considering the innermost factor $(5-\theta)$ of equation (5.5.11)

$$(5-\theta)f(x) = 5\sum_{n\geq 0} Y_n (1-x)^n - \theta \sum_{n\geq 0} Y_n (1-x)^n$$

$$= 5\sum_{n\geq 0} Y_n (1-x)^n + x\sum_{n\geq 0} Y_n (1-x)^{n-1} ,$$

or again in the original variable z,

$$(5-\theta)Y(z) = 5 \sum_{n\geq 0} Y_n z^n + (1-z) \sum_{n\geq 0} Y_n n z^{n-1}$$

$$= \sum_{n\geq 0} ((n+1)Y_{n+1} - (n-5)Y_n) z^n .$$

This means that by defining T(z) as

$$T(z) = (5-\theta)Y(z) = \sum_{n\geq 0} T_n z^n ,$$

we must have

$$T_n = (n+1)Y_{n+1} - (n-5)Y_n . \qquad (5.5.13)$$

If this process is repeated for the remaining factors $(-\theta)$ and $(-\theta-2)$ of equation (5.5.11), then by defining U(z) as

$$U(z) = (-2-\theta)T(z) = \sum_{n\geq 0} U_n z^n ,$$

we must have

$$U_n = (n+1)T_{n+1} - (n+2)T_n . \qquad (5.5.14)$$

Finally, defining $V(z)=(-\theta)U(z)= \sum_{n\geq 0} v_n z^n$ we have,

$$V_n = (n+1)U_{n+1} - nU_n \qquad (5.5.15)$$

Hence, by definition, we have

$$V(z) = (-\theta)(-\theta-2)(5-\theta)Y(z) = 6 \frac{(1-z)^3}{z^3} \sum_{n\geq 0} y_n z^n$$

$$= 6 \sum_{n\geq 3} \Delta^3 (y_n \binom{n}{3}) z^n$$

and so $$V_n = 6\Delta^3 (y_n \binom{n}{3}) . \qquad (5.5.16)$$

Thus we need to solve the following three recurrences:

$$\left.\begin{aligned} (n+1)U_{n+1} &= nU_n + V_n \\ (n+1)T_{n+1} &= (n+2)T_n + U_n \\ \text{and} \quad (n+1)Y_{n+1} &= (n-5)Y_n + T_n \end{aligned}\right\} , \text{ for } n>m . \qquad (5.5.17)$$

We are now ready to consider each of the quantities A,B,C,D,E and F in turn. If A_n is the average number of partitioning stages, then obviously $a_n=1$ and $A_n=0$ for $n\leq m$ and so from equation (5.5.6) we have,

$$A_n = \begin{cases} 1 + 2 \sum_{s=1}^{n} \frac{(s-1)(n-s)}{\binom{n}{3}} A_{s-1} & \text{for } n>m \\ 0 & \text{for } n\leq m. \end{cases} \qquad (5.5.18)$$

From equation (5.5.16) we have,

$$V_n = 6\Delta^3\left(\binom{n}{3}\right) = 6$$

and so substituting this value into the first recurrence of (5.5.17) gives,

$$(n+1)U_{n+1} = n\,U_n + 6 .$$

By telescoping this equation we have,

$$\begin{aligned} n\,U_n &= (n-1)U_{n-1} + 6 \\ &\vdots \\ (m+2)U_{m+2} &= (m+1)U_{m+1} + 6 \end{aligned}$$

which leads to,

$$\begin{aligned} (n+1)U_{n+1} &= (m+1)U_{m+1} + \sum_{k=m+1}^{n} 6 \\ &= (m+1)U_{m+1} + 6(n-m) . \end{aligned} \qquad (5.5.19)$$

If we know U_{m+1}, then we have an expression for U_{n+1} and hence U_n. Using equation (5.5.18) it is not difficult to evaluate A_{m+1}, A_{m+2} and A_{m+3} and, by substituting these values into equations (5.5.17), we may obtain U_{m+1}. In particular we have,

$$\left.\begin{aligned} T_{m+1} &= (m+2)A_{m+2} - (m-4)A_{m+1} \\ T_{m+2} &= (m+3)A_{m+3} - (m-3)A_{m+2} \\ \text{and} \qquad U_{m+1} &= (m+2)T_{m+2} - (m+3)T_{m+1} . \end{aligned}\right\} \qquad (5.5.20)$$

So, from equation (5.5.18) we have,

$$A_{m+1} = 1, \qquad A_{m+2} = 1 \text{ and } \qquad A_{m+3} = 1 + \frac{12}{(m+2)(m+3)} ,$$

and substituting these values into equations (5.5.20) gives,

$$T_{m+1} = 6, \qquad T_{m+2} = 6 + \frac{12}{(m+2)} \text{ and } \quad U_{m+1} = 6 .$$

Thus, from equation (5.5.19) we have the result,

$$(n+1)U_{n+1} = 6(m+1) + 6(n-m) = 6(n+1)$$

which leads to

$$U_n = 6 \quad .$$

Substituting this value into the second recurrence of (5.5.17) yields the equation,

$$(n+1)T_{n+1} = (n+2)T_n + 6 \ .$$

If we multiply this equation through by $\frac{1}{(n+1)(n+2)}$ we have,

$$\frac{T_{n+1}}{(n+2)} = \frac{T_n}{(n+1)} + \frac{6}{(n+1)(n+2)}$$

and treating this equation in the same way as the first recurrence we obtain,

$$\frac{T_{n+1}}{(n+2)} = \frac{T_{m+1}}{(m+2)} + 6 \sum_{k=m+1}^{n} \frac{1}{(k+1)(k+2)}$$

which leads to

$$T_n = \frac{12(n+1)}{(m+2)} - 6 \ .$$

Finally, from the third recurrence of (5.5.17) we have,

$$(n+1)A_{n+1} = (n-5)A_n + \frac{12(n+1)}{(m+2)} - 6 \ .$$

Assuming that $n \geq 6$, we may multiply this equation by $\frac{1}{6!}n(n-1)(n-2)(n-3)(n-4)$ to give,

$$\binom{n+1}{6}A_{n+1} = \binom{n}{6}A_n + \frac{12}{(m+2)}\binom{n+1}{6} - \binom{n}{5}$$

which again leads to the result,

$$\binom{n+1}{6}A_{n+1} = \binom{m+1}{6}A_{m+1} + \frac{12}{(m+2)} \sum_{k=m+1}^{n} \binom{k+1}{6} - \sum_{k=m+1}^{n} \binom{k}{5} \ .$$

Using the results obtained in Appendix A, in particular, equation (15), we may simplify this equation to obtain the result,

$$A_n = \frac{12(n+1)}{7(m+2)} - 1 + \frac{2}{7}\frac{\binom{m+1}{6}}{\binom{n}{6}} \quad . \qquad (5.5.21)$$

Now we shall consider C_n, the average number of comparisons made during partitioning. Obviously, during the first partitioning stage a comparison is made each time pointer i is increased by 1 and each time pointer j is decreased by 1. Since we start with i=2 and j=n and stop when j=i-1=s, then i is increased (s-1) times and j is

decreased (n-s) times and so C_n, the average number of comparisons made during the first partitioning stage, is (n-1).

It is obvious that $C_n=0$ when $n \leq m$ and so

$$C_n = \begin{cases} (n-1) + 2 \sum_{s=1}^{n} \frac{(s-1)(n-s)}{\binom{n}{3}} C_{s-1} & \text{for } n>m \\ 0 & \text{for } n \leq m \ . \end{cases} \qquad (5.5.22)$$

From this equation we have $C_{m+1}=m$, $C_{m+2}=m+1$ and $C_{m+3}=(m+2)+\frac{12m}{(m+2)(m+3)}$ which, when substituted into equations (5.5.17), give $T_{m+1}=7m+2$, $T_{m+3}=7m+9+\frac{12m}{(m+2)}$ and $U_{m+1}=12(m+1)$. We also have

$$V_n = 6\Delta^3\left(\binom{n}{3}(n-1)\right) = 12(2n+1) \ .$$

Using these values we solve the recurrences of (5.5.17) to obtain the results,

$$U_n = 12n \ ,$$

$$T_n = \frac{(7m-10)(n+1)}{(m+2)} + 12 + 12(n+1)(H_{n+1}-H_{m+2})$$

and finally,

$$C_n = \frac{12}{7}(n+1)(H_{n+1}-H_{m+2}) + 2 + \frac{(n+1)(37m-94)}{49(m+2)} + \frac{4}{49}(3m-1)\frac{\binom{m+1}{6}}{\binom{n}{6}} \qquad (5.5.23)$$

where H_n, the n^{th} harmonic number, is defined as

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots . \frac{1}{n} \ .$$

If we now consider B_n, the average number of exchanges during partitioning, then, as with C_n, $B_n=0$ for $n \leq m$. Now during the first partitioning stage, if s is the partitioning element, the number of exchanges will be the number of elements among $a_3,a_4,\ldots a_s$ that are greater than s. Averaging over all permutations of {1,2,...n} we find that the average number of exchanges when s is the partitioning element is,

$$\sum_{t=0}^{s-2} t \frac{\binom{n-s-1}{t}\binom{s-2}{s-2-t}}{\binom{n-3}{s-2}} = \frac{(n-s-1)(s-2)}{(n-3)} \ ,$$

and averaging this over all partitioning elements s, we find that,

$$b_n = \sum_{s=1}^{n} \frac{(s-1)(n-s)}{\binom{n}{3}} \cdot \frac{(n-s-1)(s-2)}{(n-3)}$$

$$= \frac{(n-4)}{5} \quad . \tag{5.5.24}$$

Thus we have,

$$B_n = \begin{cases} \frac{(n-4)}{5} + 2 \sum_{s=1}^{n} \frac{(n-s)(s-1)}{\binom{n}{3}} \cdot B_{s-1} & , \text{ for } n>m \\ 0 & , \text{ for } n \leq m, \end{cases} \tag{5.5.25}$$

but this relationship is a linear combination of A_n and C_n and so,

$$B_n = \frac{1}{5}(C_n - 3A_n)$$

$$= \frac{12}{35}(n+1)(H_{n+1} - H_{m+2}) + 1 + \frac{37}{245}(n+1) - \frac{12(n+1)}{7(m+2)} + \frac{2}{245}(6m-23)\frac{\binom{m+1}{6}}{\binom{n}{6}} \cdot \tag{5.5.26}$$

Next we have F_n which is the average number of linear insertion stages. A linear insertion stage occurs when a subset of at most m elements is created, and this happens during the first partitioning stage when s has the value $3,4,\ldots(m+1)$ or $(n-m),(n-m+1),\ldots(n-2)$. Thus we have,

$$f_n = \sum_{s=3}^{m+1} \frac{(s-1)(n-s)}{\binom{n}{3}} + \sum_{s=n-m}^{n-2} \frac{(s-1)(n-s)}{\binom{n}{3}}$$

$$= 2 \sum_{s=3}^{m+1} \frac{(s-1)(n-s)}{\binom{n}{3}}$$

$$= 6\frac{\binom{m+1}{2}}{\binom{n}{2}} - 4\frac{\binom{m+1}{3}}{\binom{n}{3}} - \frac{6}{\binom{n}{2}} \quad , \tag{5.5.27}$$

which gives,

$$F_n = \begin{cases} 6\frac{\binom{m+1}{2}}{\binom{n}{2}} - 4\frac{\binom{m+1}{3}}{\binom{n}{3}} - \frac{6}{\binom{n}{2}} + 2\sum_{s=1}^{n} \frac{(s-1)(n-s)}{\binom{n}{3}} \cdot F_{s-1} & , \text{ for } n>m \\ 0 & , \text{ for } n \leq m \ . \end{cases} \tag{5.5.28}$$

Considering the components of F_n one at a time we have

$$F'_n = \begin{cases} \frac{1}{\binom{n}{2}} + 2 \sum_{s=1}^{n} \frac{(s-1)(n-s)}{\binom{n}{3}} F'_{s-1} & , \text{ for } n>m \\ 0 & , \text{ for } n \leq m \ , \end{cases}$$

which proceeding as before yields the result

$$F'_n = \frac{8(n+1)}{7m(m+1)(m+2)} + \frac{6}{7m(m+1)} \frac{\binom{m+1}{6}}{\binom{n}{6}} ,$$

and

$$F''_n = \begin{cases} \frac{1}{\binom{n}{3}} + 2 \sum_{s=1}^{n} \frac{(s-1)(n-s)}{\binom{n}{3}} F''_{s-1} & , \text{ for } n>m \\ 0 & , \text{ for } n \leq m, \end{cases}$$

which leads to the result,

$$F''_n = \frac{18(n+1)}{7(m-1)m(m+1)(m+2)} + \frac{24}{7(m-1)m(m+1)} \cdot \frac{\binom{m+1}{6}}{\binom{n}{6}} .$$

Now F_n is simply a linear of combination of F'_n and F''_n and so,

$$F_n = 6[\binom{m+1}{2} - 1]F'_n - 4\binom{m+1}{3}F''_n$$

$$= \frac{12}{7}\left(1 - \frac{4}{m(m+1)}\right)\frac{(n+1)}{(m+2)} + \frac{2}{7}\left(1 - \frac{18}{m(m+1)}\right) \frac{\binom{m+1}{6}}{\binom{n}{6}} .$$

The remaining two quantities D_n and E_n are the contribution made by linear insertion. First of all we must consider small subsets that are sorted by linear insertion.

With each permutation $a_1, a_2, \ldots, a_n$ of {1,2,3....n} we associate an inversion table $I_1, I_2, I_3, \ldots I_n$ such that I_i is the number of elements to the left of a_i that are greater than a_i. I_1 is always 0 and we must have

$$0 \leq I_2 \leq 1, 0 \leq I_3 \leq 2 \ldots\ldots\ldots 0 \leq I_n \leq n-1 .$$

The average number of insertions, D_n, is the number of elements with at least one element to its left greater than itself which is also the number of non-zero entries in the inversion table. The probability that $I_i \neq 0$ is $1-\frac{1}{i}$ for all i and thus

$$D_n = (1-1) + (1-\tfrac{1}{2}) + \ldots\ldots (1-\tfrac{1}{n})$$

$$= n - H_n . \qquad (5.5.29)$$

The second quantity E_n, the number of moves made during linear insertion is equal to the sum of the entries in the inversion table

since each element has to be moved past every element to its left which is greater than itself. Therefore we have

$$E_n = I_1+I_2+\ldots.I_n \ ,$$

this total being the number of inversions of the permutation, an inversion being a pair (a_i,a_j) where $i<j$ and $a_i>a_j$. The minimum value of E_n is 0 and the maximum is clearly $\binom{n}{2}$ when $I_i=i-1$ for $i=1(1)n$.

We notice that if a permutation $a_1,a_2,a_3,\ldots,a_n$ has k inversions, then $a_n,a_{n-1},\ldots.,a_1$ will have $\binom{n}{2}-k$ inversions. So if the probability that a permutation of $\{1,2,\ldots.n\}$ has exactly k inversions is e_{nk} and $k'=\binom{n}{2}-k$ then,

$$e_{nk} = e_{nk'} \ .$$

So the average number of inversions E_n is

$$E_n = \sum_k k e_{nk} = \sum_k (\tbinom{n}{2}-k)e_{nk'}$$

$$= \sum_k (\tbinom{n}{2}-k)e_{nk} \ ,$$

and so we have

$$2E_n = \sum_k (k + \tbinom{n}{2}-k)e_{nk}$$

$$= \tbinom{n}{2} \sum_k e_{nk} \ .$$

Since $\sum_k e_{nk}=1$, we must have

$$2E_n = \tbinom{n}{2} \ ,$$

$$\Rightarrow \qquad E_n = \frac{n(n-1)}{4} \ . \qquad (5.5.30)$$

Returning to the original problem, D_n is the average number of non-zero entries in the inversion table after partitioning and E_n is the average number of inversions in the permutation after partitioning. Now the partitioning process places s into its final position and so I_s becomes zero. Furthermore, if an inversion table entry in either subset is non-zero it must be because there is a larger

element to its left in that subset. Thus the number of non-zero entries in the inversion table for the whole set is the sum of the non-zero entries in the inversion tables of the subsets. Similarly the sum of the inversion table entries for the whole set is the sum of the sums of inversion table entries for the subsets. Thus we have the two relationships:

$$D_n = \begin{cases} 2\sum_{s=1}^{n} \frac{(n-s)(s-1)}{\binom{n}{3}} D_{s-1} & \text{for } n>m \\ n-H_n & \text{for } n\leq m \end{cases} \qquad (5.5.31)$$

and

$$E_n = \begin{cases} 2\sum_{s=1}^{n} \frac{(n-s)(s-1)}{\binom{n}{3}} E_{s-1} & \text{for } n>m \\ \frac{n(n-1)}{4} & \text{for } n\leq m\ . \end{cases} \qquad (5.5.32)$$

We may now proceed in exactly the same way as we did for the other quantities to obtain the results,

$$D_n = (n+1) - \frac{4(n+1)}{7(m+2)}(3H_{m+1}+1) + \frac{1}{7}(\frac{5}{3} - 2H_{m+1})\frac{\binom{m+1}{6}}{\binom{n}{6}} \qquad (5.5.33)$$

and

$$E_n = \frac{(6m-17)}{35}(n+1) + \frac{6(n+1)}{7(m+2)} - \frac{(3m^2+15m-2)}{140}\frac{\binom{m+1}{6}}{\binom{n}{6}}\ . \qquad (5.5.34)$$

For large values of n we may ignore the terms with a denominator of $\binom{n}{6}$ and so we have the six formulae:-

$$A_n = \frac{12(n+1)}{7(m+2)} - 1\ ,$$

$$B_n = \frac{12}{35}(n+1)(H_{n+1}-H_{m+2}) + 1 + \frac{37}{245}(n+1) - \frac{12(n+1)}{7(m+2)}\ ,$$

$$C_n = \frac{12}{7}(n+1)(H_{n+1}-H_{m+2}) + 2 + (n+1)\frac{(37m-94)}{49(m+2)}\ ,$$

$$D_n = (n+1) - \frac{4(n+1)}{7(m+2)}(3H_{m+1}+1)\ ,$$

$$E_n = \frac{(n+1)}{35}(6m-17) + \frac{6(n+1)}{7(m+2)}$$

and

$$F_n = \frac{12}{7}\left(1 - \frac{4}{m(m+1)}\right)\frac{(n+1)}{(m+2)}\ .$$

Since the frequency with which each statement in the Parallel Quicksort program is executed is dependent on these quantities, then by estimating the time that each statement takes to be executed we obtain a formula for $\tilde{t}$ of the form,

$$\tilde{t} = aA_n + bB_n + cC_n + dD_n + eE_n + fF_n + gn \ . \qquad (5.5.35)$$

The values of the coefficients a,b,c,d,e,f and g will vary from computer to computer but, by applying the statement times described in Appendix A to program (7) (Appendix B), we obtain the following result,

$$\begin{aligned}\tilde{t} &= 184A_n + 30B_n + 16C_n + 38D_n + 32E_n + 53F_n + 28n \\ &= \frac{264}{7}(n+1)H_{n+1} - 150 + (n+1)\left\{\frac{16432}{245} + \frac{2140}{7(m+2)} - \frac{264}{7}H_{m+2} - \frac{456}{7(m+2)}H_{m+1}\right. \\ &\qquad \left. + \frac{192}{35}m - \frac{2544}{7m(m+1)(m+2)}\right\} \text{ units,}\end{aligned} \qquad (5.5.36)$$

where 1 unit is approximately a machine instruction time.

We now wish to estimate the times t_1 and $\tilde{t}_1$, the parallel and sequential run times of phase 1. Clearly the average size of the left subset produced by the first partitioning stage is,

$$\sum_{s=1}^{n} (s-1)\,\frac{(s-1)(n-s)}{\binom{n}{3}} = \frac{(n-1)}{2}$$

Similarly the average size of the right subset is $\frac{(n-1)}{2}$. The partitioning of these two subsets produces four subsets of average size

$$(\frac{(n-1)}{2} - 1)/2 \ .$$

During the initial phase, this process is repeated until at least p subsets have been produced. If the concurrent partitioning of the two subsets of size (n-1)/2 and similarly the four subsets produced by these partitioning steps is called a parallel partitioning

stage, then to produce p subsets we require j parallel partitioning stages where,

$$j = \lceil \log_2 p \rceil \quad , \qquad (\text{i.e. } 2^j \geq p) \ .$$

Obviously, if q_i is the average size of the subsets at the i^{th} parallel partitioning stage, then

$$q_1 = n$$

$$q_i = (q_{i-1}-1)/2 \qquad \text{for} \quad i=2,3,\ldots,j,$$

from which we obtain the formula,

$$q_i = (n-2^{i-1}+1)/2^{i-1} \quad \text{for } i=1,2,\ldots,j. \tag{5.5.37}$$

From the previous analysis we know that the contribution of the first partitioning stage is dependent on the quantities,

$$a_n = 1, \quad b_n = (n-4)/5 \text{ and } \quad c_n = (n-1),$$

so for a subset of size q_i we have

$$\left.\begin{aligned} a_{q_i} &= 1 \\ b_{q_i} &= (q_i-4)/5 \\ \text{and} \qquad c_{q_i} &= (q_i-1) \ . \end{aligned}\right\} \tag{5.5.38}$$

If the time required by each partitioning step is α_i, then, treating α_i in the same way as we treated $\tilde{t}$, we have

$$\alpha_i = 212a_{q_i} + 30b_{q_i} + 16c_{q_i} \ , \tag{5.5.39}$$

and substituting in the values from equations (5.5.38) we obtain,

$$\alpha_i = 172 + 22q_i \qquad \text{for } i=1(1)j. \tag{5.5.40}$$

Clearly the average value of t_1 is,

$$t_1 = \sum_{i=1}^{j} \alpha_i \ ,$$

and so, using equations (5.5.37) and (5.5.40), we have

$$t_1 = 150j + 44(n+1)(1-2^{-j}) \ . \tag{5.5.41}$$

Since, during the i^{th} parallel partitioning stage, there are 2^{i-1} subsets being partitioned concurrently, it is clear that,

$$\tilde{t}_1 = \sum_{i=1}^{j} 2^{i-1}\alpha_i \ ,$$

which reduces to,

$$\tilde{t}_1 = 150(2^j-1) + 22(n+1)j \ . \qquad (5.5.42)$$

Thus, since t_3 is small enough to be ignored, we obtain, from equations (5.5.1) and (5.5.2), the formula,

$$T_p = t_1 + (\tilde{t}-\tilde{t}_1)/p \quad , \qquad (5.5.43)$$

where the quantities $\tilde{t}$, t_1 and $\tilde{t}_1$ may be obtained from equations (5.5.36), (5.5.41) and (5.5.42) respectively.

We now wish to find the optimum value of m, the best choice of subset size for which it is more efficient to sort by linear insertion. Obviously this is achieved by minimising T_p with respect to m; in particular we must minimise the function

$$g(m) = \frac{16432}{245} + \frac{2140}{7(m+2)} - \frac{264}{7}H_{m+2} - \frac{456}{7(m+2)}H_{m+1} + \frac{192}{35}m - \frac{2544}{7m(m+1)(m+2)}$$

with respect to m. (5.5.44)

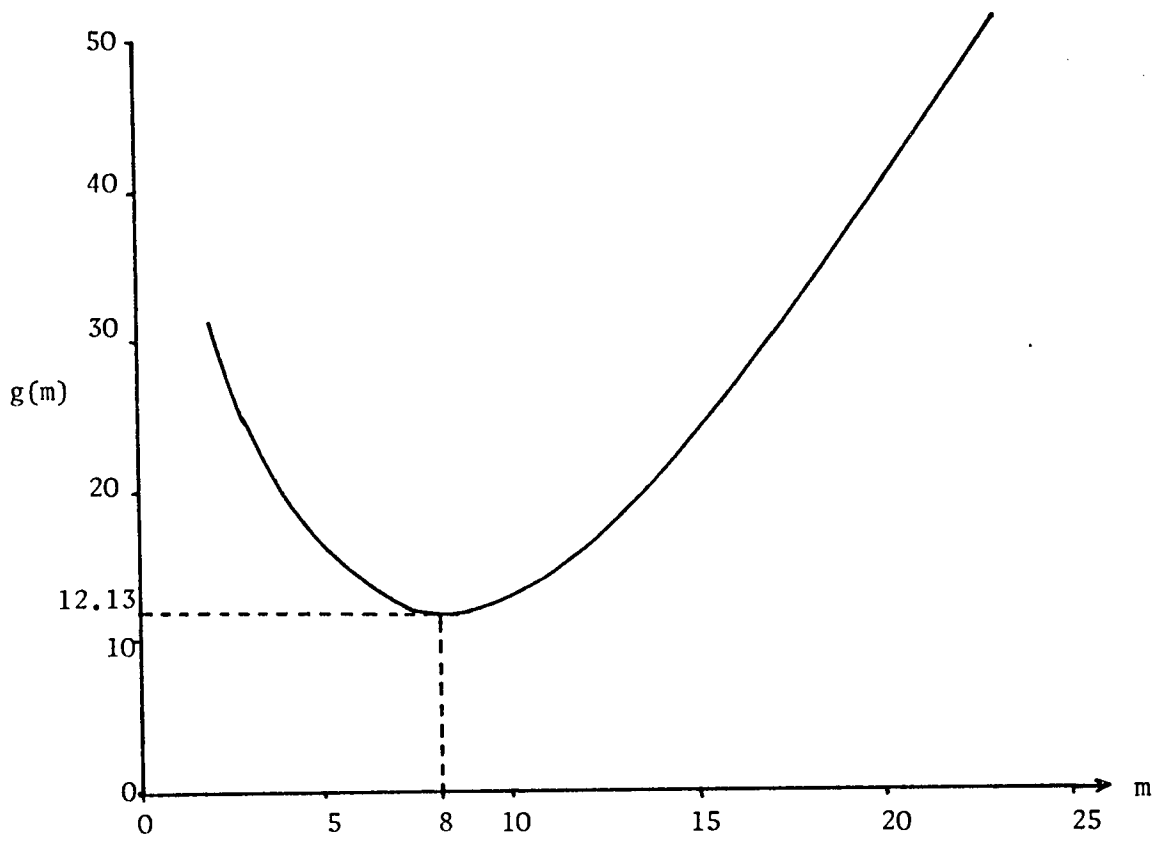


FIGURE 5.4

In Figure 5.4, values of g(m) are plotted against m. From the graph we can see that the optimum value of m is 8 but clearly the choice of m is not critical and so any choice of m between 5 and 12 is viable.

To complete the analysis of the Parallel Quicksort algorithm we shall investigate the effect that p, the number of processors, has on the performance of the algorithm. To achieve this we must refer to the quantities Speed-up and Efficiency defined in Chapter 2 and the Performance Factor defined by (4.2.6).

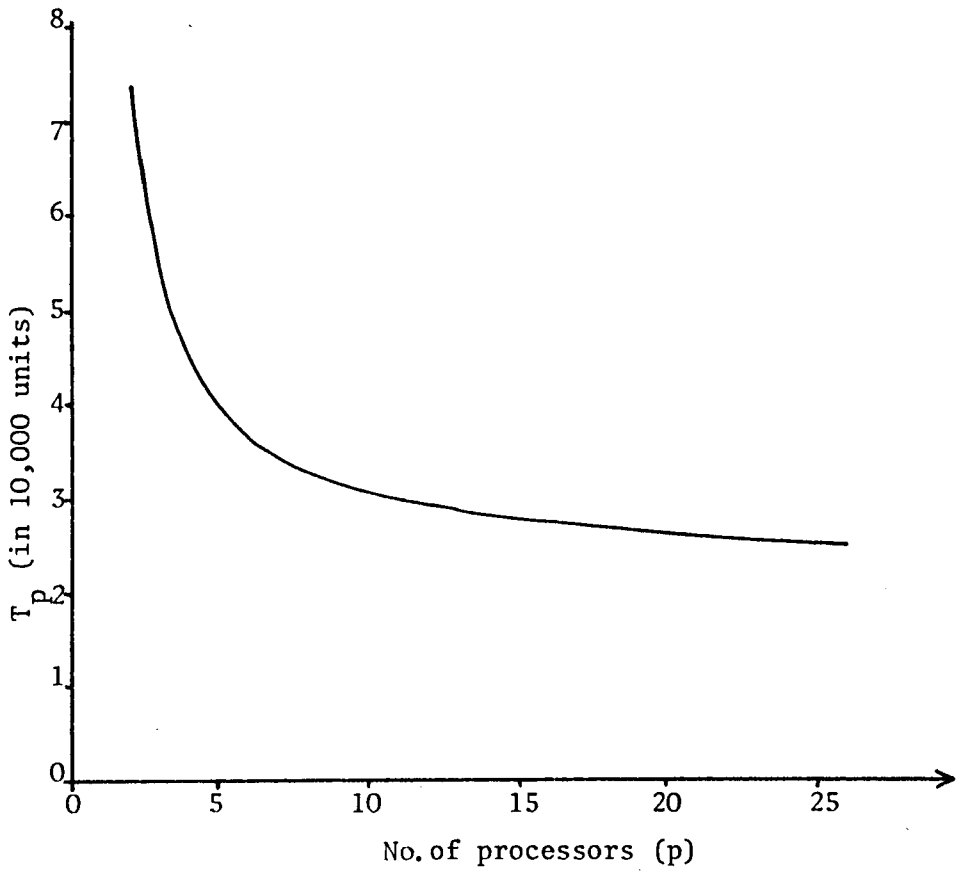


FIGURE 5.5

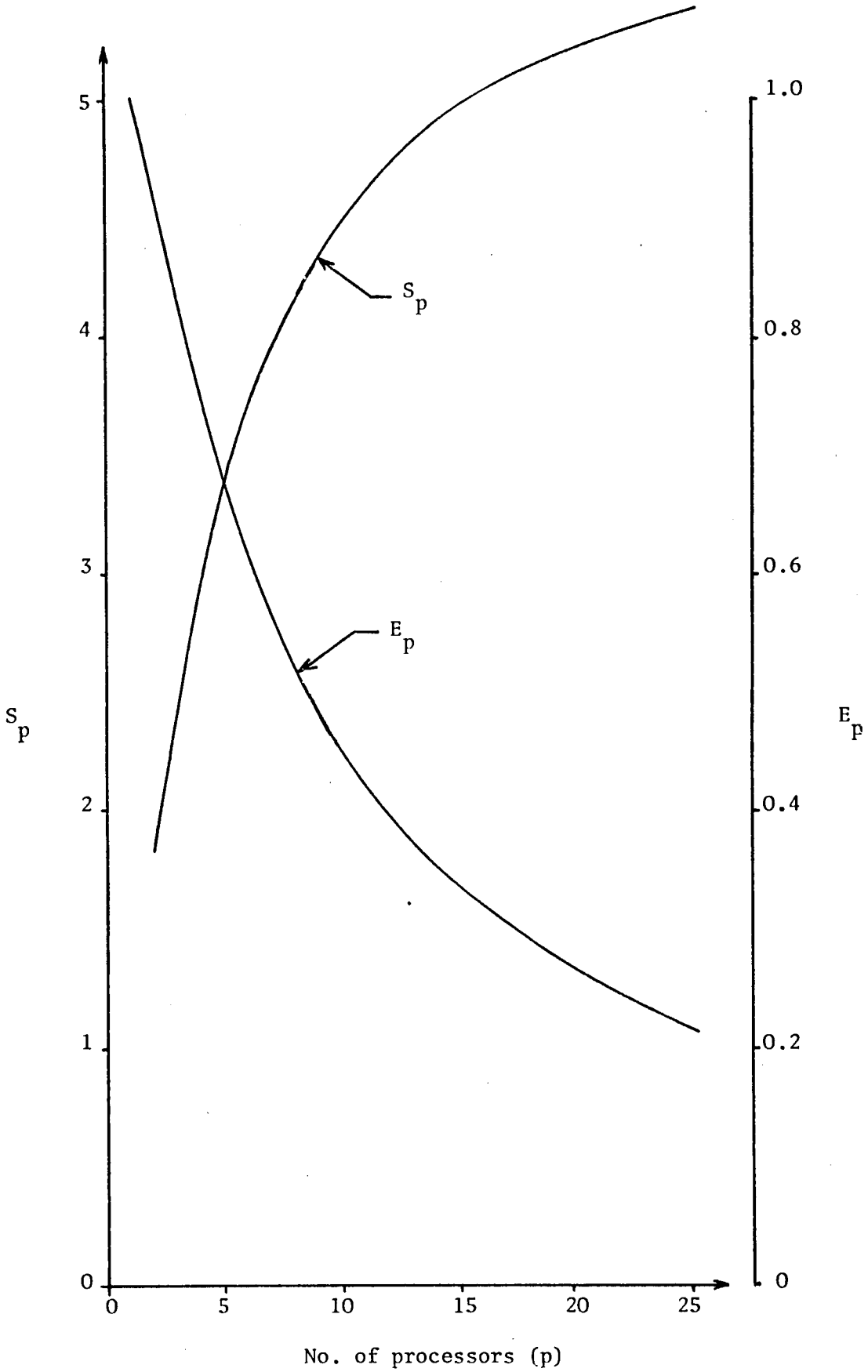

5
4
3
2
1
0
1.0
0.8
0.6
0.4
0.2
0
S_p
E_p
S_p
E_p
0
5
10
15
20
25
No. of processors (p)

FIGURE 5.6

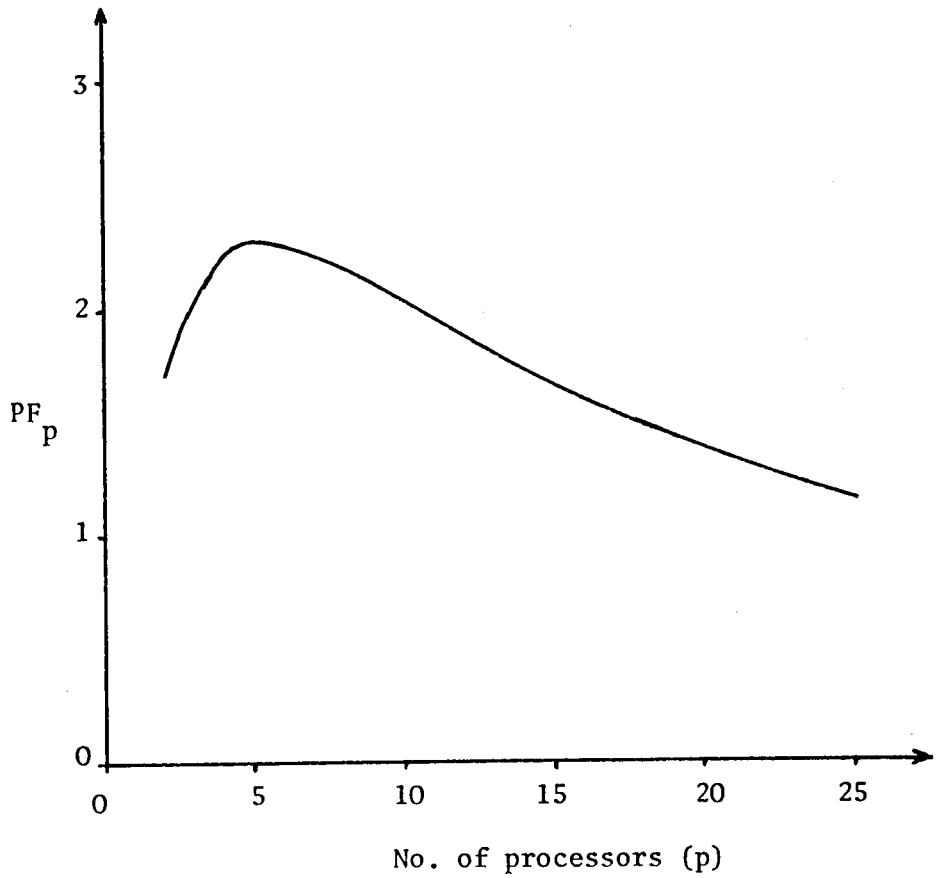


FIGURE 5.7

The series of Figures (5.5)-(5.7) plot the quantities time (T_p), Speed-up (S_p) and Efficiency (E_p), and Performance Factor (PF_p) against p respectively for n=500. The graphs are essentially the same for m=5,6,...15 which demonstrates that the choice of m is not critical.

Closer examination of these graphs reveals that little improvement is achieved by increasing p above 15. We can also see that for p>15 the Speed-up settles at approximately 5 while the Efficiency steadily decreases. Considering the Performance Factor in Figure 5.7, we see that the optimum value of p is 5. However, it is also clear that when p is 4,5 or 6 the performance of Parallel Quicksort is very good.

Thus we conclude this analysis by observing that for n=500 the best choice of m lies between 5 and 12 while the optimum number of processors is 5. As n increases we would expect the optimum value of

p to increase very slowly. However the best choice of m remains constant for all values of n.

It must be stressed that a more accurate assessment of the best values of p and m for a particular computer may be obtained by using the actual values of a,b,c,d,e,f and g in equation (5.5.35) for that particular computer. It is also useful to remember that the time overhead incurred by memory contention (excluded from this analysis) has a less damaging effect for smaller values of p and so it is better to underestimate the optimum value of p.

5.6 SIMULATION OF THE PARALLEL QUICKSORT METHOD

In the absence of a suitable parallel computer to test the Parallel Quicksort Method, the method was simulated and the results compared with those of section 5.5.

For the purposes of this event-orientated simulation we define the following variables:-

an array R[a,b], containing the information which is obtained during the sorting process,

where a = the subset number

and b = an integer in the range 1 to 5.

R[a,1] and R[a,2] are pointers to the left and right subsets respectively, produced by the partitioning of subset a.

R[a,3] and R[a,4] indicate the lower and upper limits respectively of subset a.

R[a,5] is an estimate of the time required to partition or perform linear insertion on subset a.

The time R[a,5] may be obtained for subset a by adding the time it takes to execute a statement to a running total each time that

statement is used. When a subset is created, it is given a number and its limits may be recorded in the array R in the appropriate positions. At the same time, a pointer to that subset may be placed in R[a,1] or R[a,2] accordingly, where subset a is the subset from which the new subset has been created.

The simulated model of the parallel computer is represented by the following variables:-

p = the number of processors,

T = a clock,

U[i] (i=1,2,...,p) = stack of processors in use,

A[i] (i=1,2,...,p) = stack of processors available,

S[a,b] = processor b,

where S[1,b] = the length of time before which processor b becomes available,

and S[2,b] = the number of the subset currently being processed by processor b,

Q[j] = a queue of subsets that are waiting to be processed,

LU = number of processors in use,

LA = number of processors available,

and LQ = number of subsets in queue.

When a subset of size 0 or 1 is created, the corresponding pointer is set to zero, and when a subset is $\leq m$ in size (i.e., when it is sorted by linear insertion), both pointers are set to zero.

As soon as all of the information concerned with the application of the algorithm has been recorded in array R, we proceed in the following manner. Initially the queue Q is empty and all p processors are available. The partitioning of the complete set is assigned to processor 1 by removing processor 1 from the processors available stack,

placing it on the processors in use stack and then setting S[1,1]=R[1,5] and S[2,1]=1.

Since an event occurs when a processor 'in use' becomes 'available' the simulation proceeds by searching the processors in use to find the next processor to become available i.e., the processor in use with the smallest value S[1,b]. The clock is then advanced by this amount of time and all the S[1,b]'s of the processors in use decreased by the same amount. Then, for each processor in use with S[1,b]=0, the appropriate event is performed and the procedure is repeated. The simulation is terminated when all processors are available and the queue Q is empty.

There are five different situations that may occur which lead to different events which are described as follows:-

Event 1

A small subset has been sorted by linear insertion, so no new subsets have been created and the queue Q of unprocessed subsets is empty. The processor that has sorted that subset is therefore removed from the stack of processors in use to the stack of processors available.

Event 2

Again a small subset has been sorted by linear insertion but now the queue Q is not empty. The subset at the head of the queue is immediately assigned to the processor that has just become available.

Event 3

A subset has been partitioned to create only one new subset (i.e., the other new subset is of size 0 or 1). The new subset is immediately assigned to the processor that has become available.

Event 4

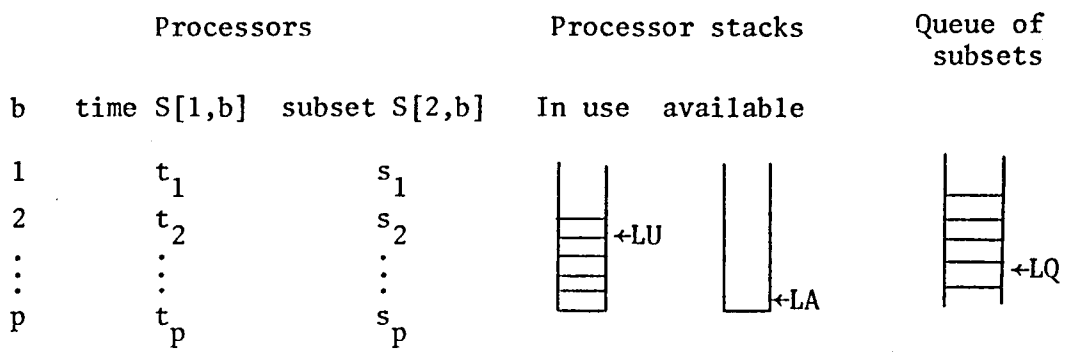
A subset has been partitioned creating two new subsets and the processors available stack is not empty. One of the subsets is assigned

to the processor that has become available and the other to the processor at the top of the processors available stack. That processor is removed from the processors available stack to the processors in use stack.

Event 5

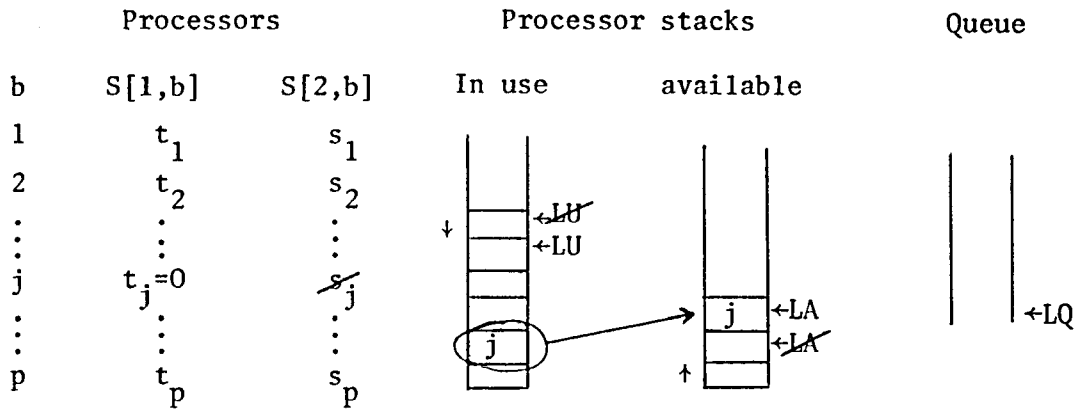
A subset has been partitioned to create two new subsets and no processors are available. One of the subsets is assigned to the processor that has become available and the other is placed in the queue Q.

The current state of the simulated parallel computer may be represented by the following diagram.



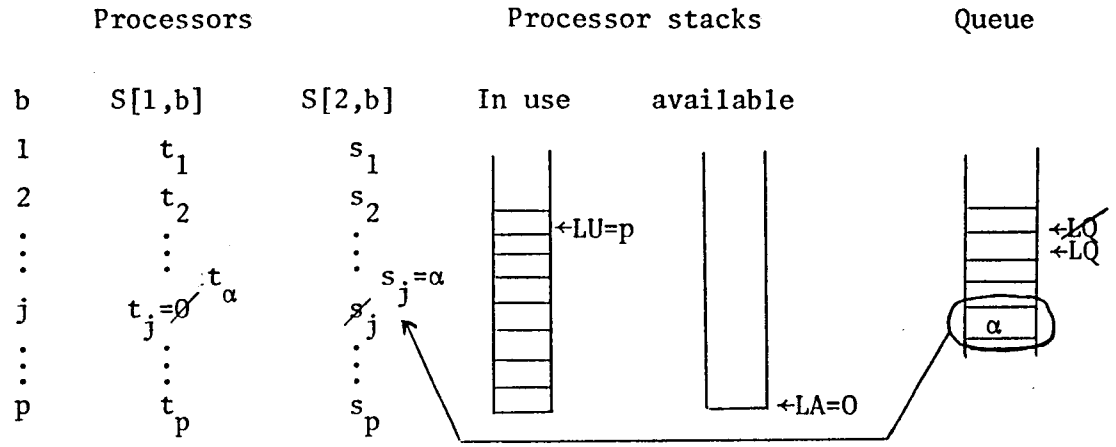
Since all of the processors are in use, the processors available stack is empty. If this representation of a parallel computer is used, we can illustrate each of the five events. In each of the examples, processor j has become available.

Event 1



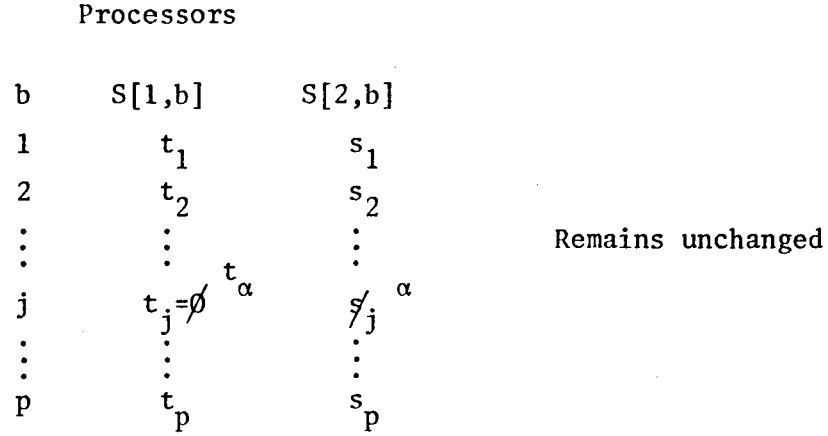
The stack of available processors may initially be empty.

Event 2



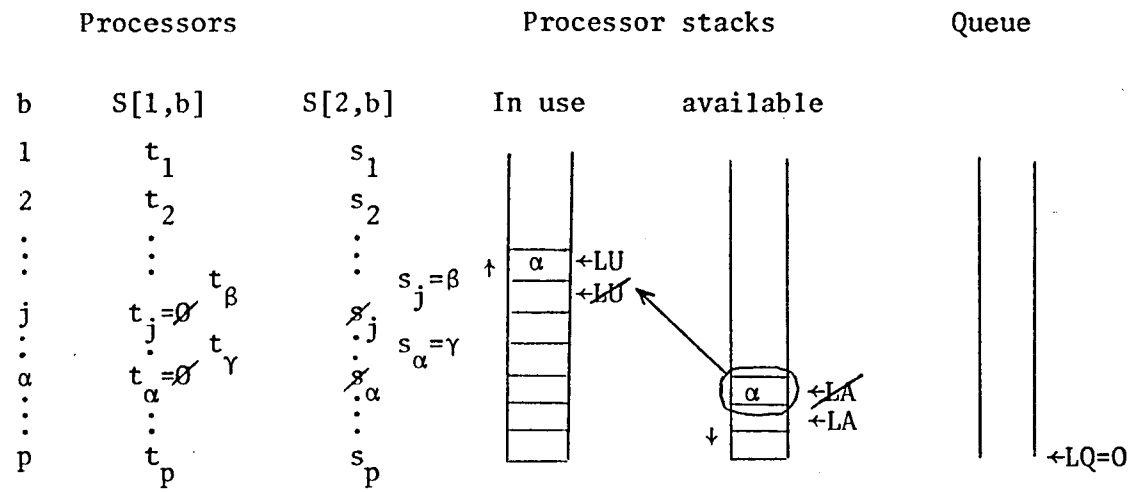
Since the queue is not empty, all processors must be in use.

Event 3



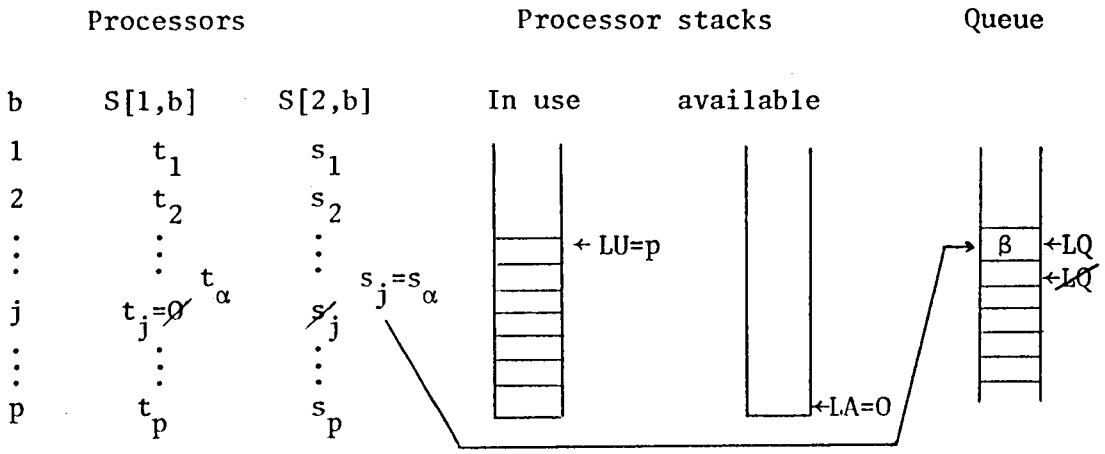
Only one new subset is created, call it subset α, and, since it is assigned to processor j, the processor stacks and queue are unchanged.

Event 4



Two new subsets are created numbered β and γ. Since some processors are available the queue must be empty.

Event 5



Two new subsets α and β have been created, α being assigned to processor j and β being placed in the queue. Since the queue is not empty, all of the processors must be in use.

5.7 RESULTS AND CONCLUSIONS

In order to test the Parallel Quicksort method using the simulation described in section 5.6, sets of pseudo-random numbers were obtained using the NAG library random number generator routines (NAG, 1976). The routines involved include G05AAA, G05ABA and G05BAA.

The routine G05AAA returns pseudo-random numbers from a uniform distribution on the range (0,1) by generating two multiplicative congruential sequences

$$\left.\begin{aligned} x_{1,r+1} &= (b_1 x_{1,r}) \bmod m \\ \text{and} \qquad x_{2,r+1} &= (b_2 x_{2,r}) \bmod m \quad , \end{aligned}\right\} \tag{5.7.1}$$

where (z) mod m is the remainder left when z is divided by m. A sequence of pseudo-random numbers x_{r+1}, is then formed using,

$$x_{r+1} = (x_{1,r+1} + x_{2,r+1}) \bmod m \ , \tag{5.7.2}$$

which are then scaled to produce the required sequence. The values of the constants are machine orientated, being related to the word size of the ICL 1900, and are $m=2^{46}$, $b_1=3^{15}$, $b_2=5^9$, and $x_{1,0}=x_{2,0}=1234567$.

In order to obtain pseudo-random numbers from a uniform distribution on the range (a,b), the sequence x_{r+1} generated by routine G05AAA was scaled by routine G05ABA using the transformation,

$$y_{r+1} = (b-a)x_{r+1} + a \quad . \tag{5.7.3}$$

Finally, to generate different sets of pseudo-random numbers, routine G05BAA was used to initialise the routine G05AAA by setting the parameter $x_{2,0}$ to a value derived from a parameter x of routine G05BAA.

The sets of random numbers used to test the Parallel Quicksort method were integers lying in the range (0,100,000) and were obtained by setting parameters a and b of routine G05ABA to 0 and 100,000 respectively, each number y_{r+1} being rounded down to the greatest integer less than y_{r+1}.

In standard Quicksort, the maximum recursive depth of the algorithm is minimised by sorting the smaller of the two subsets produced by partitioning first. To see if this is a desirable feature for Parallel Quicksort, the initial tests were performed, also sorting the smaller subset first.

Initially the sorting of 10 different sets of 500 random numbers by Parallel Quicksort on a p processor computer was simulated. The number of processors p was varied from 2 up to 16 and m, the size of the largest subset for which linear insertion is used, was given the values 5,10 and 15. The results, including the average run-time, Speed-up, Efficiency and Performance Factor of the method when m=10 are recorded in Table 5.1 in the columns headed by the letter A.

p	Average T_p		Speed-up		Efficiency		Performance Factor	
	A	B	A	B	A	B	A	B
2	71635	71367	1.842	1.849	0.921	0.925	1.697	1.710
3	53487	53098	2.467	2.485	0.822	0.828	2.029	2.059
4	45743	44867	2.885	2.941	0.721	0.735	2.081	2.163
5	42066	40920	3.137	3.225	0.627	0.645	1.968	2.080
6	39851	39038	3.312	3.381	0.552	0.563	1.828	1.905
7	38662	38202	3.413	3.454	0.488	0.494	1.664	1.705
8	38041	37671	3.469	3.503	0.434	0.438	1.504	1.534
9	37736	37453	3.497	3.524	0.389	0.392	1.359	1.380
10	37518	37306	3.517	3.537	0.352	0.354	1.237	1.251
11	37422	37194	3.526	3.548	0.321	0.323	1.131	1.144
12	37274	37125	3.540	3.555	0.295	0.296	1.045	1.053
13	37243	37088	3.543	3.558	0.273	0.274	0.966	0.974
14	37216	37088	3.546	3.558	0.253	0.254	0.898	0.904
15	37157	37088	3.552	3.558	0.237	0.237	0.841	0.844
16	37151	37088	3.552	3.558	0.222	0.222	0.789	0.791

Results of the simulation of Parallel Quicksort on a p processor computer with m=10

TABLE 5.1

These tests were then repeated, using the same sets of random numbers but sorting the larger of the two subsets produced by partitioning first. The results of these tests when m=10 are also recorded in Table 5.1 under the columns headed by the letter B. Similar results were also obtained for m=5 and 15.

From Table 5.1 we observe that the method is faster when the larger subset produced by partitioning is sorted first and a closer examination of the simulations reveals that this is because the period during the second phase of the method, when not all p processors are in use, has been reduced.

It can also be seen, especially from the second series of tests, that when p>12 the Speed-up remains constant. Again, from a closer examination of the simulations, we see that this is because all p processors are never in use concurrently at any point during the execution of the algorithm.

Finally we observe from the performance factor that the optimum value of p lies in the range 3 to 6.

So the first series of tests has revealed that it is better to sort the larger subset produced by partitioning first and that the optimum value of p lies in the range 3 to 6. In the next series of tests we attempt to optimise the values of p and m.

Accordingly, in the next series of tests, the average run time of the method is found from a sample of 20 different sets of 500 random numbers. The simulation of Parallel Quicksort is carried out with m varying from 5 to 15 and p varying from 3 to 6. The results from these tests for m=5,6,7.....10 are recorded in Table 5.2.

It is clear that for all values of m the optimum number of processors is 4 but 5 is also a very good choice. However since the effects of store clashing is less for smaller values of p, it is better to underestimate the value of p.

From the results we see that the optimum value of m is 6, but again the choice of m is not critical and any choice of m in the range 5 to 10 is equally as good.

Also included in Table 5.2 are the theoretical run times of the algorithm so that they may be compared with the results obtained from the simulation. Obviously there is reasonable agreement between the two sets of results.

p	Average T_p	Theoretical T_p	Speed-up	Efficiency	Performance Factor
m=5					
3	52100	54728	2.496	0.832	2.076
4	43892	45254	2.962	0.741	2.194
5	39704	40151	3.275	0.655	2.145
6	37627	36749	3.456	0.576	1.990
m=6					
3	51894	54335	2.493	0.831	2.071
4	43741	44960	2.957	0.739	2.186
5	39618	39916	3.265	0.653	2.132
6	37579	36553	3.442	0.574	1.975
m=7					
3	51901	54143	2.491	0.830	2.068
4	43798	44816	2.952	0.738	2.178
5	39702	39800	3.256	0.651	2.121
6	37579	36457	3.440	0.573	1.972
m=8					
3	52085	54106	2.493	0.831	2.071
4	43951	44788	2.954	0.739	2.182
5	39808	39778	3.262	0.652	2.128
6	37684	36438	3.445	0.574	1.978
m=9					
3	52328	54190	2.493	0.831	2.071
4	44104	44850	2.957	0.739	2.186
5	39928	39828	3.267	0.653	2.134
6	37841	36480	3.447	0.574	1.980
m=10					
3	52725	54368	2.492	0.831	2.070
4	44351	44985	2.963	0.741	2.194
5	40138	39935	3.274	0.655	2.143
6	37990	36569	3.459	0.576	1.994

Results of the simulation of Parallel Quicksort for different values of m

TABLE 5.2

If we now compare the optimum values of p and m obtained by simulation and by the analysis in section 5.5, we see that the actual

optimum values are not exactly in agreement.

From the simulation results the optimum value of p is 4 but from the analysis it is 5. However, since it is better to underestimate p we conclude that the optimum number of processors is 4 (when n=500)..

Regarding the optimum value of m, we see that although the optimum values from the simulation results and the analysis are not the same, the ranges for m are. Since the actual value of m is not critical it can therefore be concluded that m should lie in the range 5 to 10.

Finally, we repeat the observation that the best value of m will be constant for all values of n but it can be expected that as n increases, the optimum value of p will increase very slowly. This fact can be confirmed by the observation that, when n is doubled, the optimum value of p is only increased by 1. From these results it seems that a functional relationship between p and n of the form $p = a.\log n + b$ can be established.

Time rather than space analysis has been included in this chapter as it is intended to assess the potential time improvement of the algorithm which is the basis of parallel computing, whereas space saving is not of primary importance.

CHAPTER 6

SUCCESSIVE OVER-RELAXATION - A PARALLEL APPROACH

6.1 INTRODUCTION

Many problems occurring in science that involve the rates of change with respect to two or more variables produce one or a set of partial differential equations when formulated mathematically. A case that occurs more frequently than any other is the two dimensional second order equation

$$a\frac{\partial^2\phi}{\partial x^2} + b\frac{\partial^2\phi}{\partial x\partial y} + c\frac{\partial^2\phi}{\partial y^2} + d\frac{\partial\phi}{\partial x} + e\frac{\partial\phi}{\partial y} + f\phi + g = 0 \ , \qquad (6.1.1)$$

where a,b,c,d,e,f and g may be functions of the independent variables x and y and of the dependent variable ϕ. This equation is said to be elliptic when $(b^2-4ac)<0$, parabolic when $(b^2-4ac)=0$ or hyperbolic when $(b^2-4ac)>0$.

In this chapter we shall investigate the solution of elliptic partial differential equations which, in general, are associated with steady state situations such as the steady flow of heat or electricity in homogeneous conductors. In particular, we shall consider the solution of Laplace's equation, defined as,

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0 \ , \qquad (6.1.2)$$

over a closed region with the Dirichlet boundary conditions,

$$\left.\begin{aligned} \phi(x,0) = \phi(0,y) = \phi(1,y) = 0 \\ \text{and} \qquad \phi(x,1) = 1 \end{aligned}\right\} \ . \qquad (6.1.3)$$

At present, only a limited number of elliptic partial differential equations have been solved analytically and even for those, the analytical solution is often extremely laborious to evaluate. Elliptic equations are therefore usually solved by numerical approximation methods such as finite-difference methods.

In finite-difference methods, a system of rectangular meshes is formed over the region of integration of the elliptic equation by two

sets of equally spaced lines, one set parallel to the x axis and the other parallel to the y axis. This is illustrated for Laplace's equation with Dirichlet boundary conditions in Figure 6.1 where each set of parallel lines are at a distance h apart. At each mesh point, i.e., the points of intersection of the parallel lines, the partial differential equation is approximated by replacing it by a finite-difference equation. The finite-difference equation defines each mesh point in terms of the neighbouring mesh points and, when applied to all of the mesh points, produces a large system of algebraic equations.

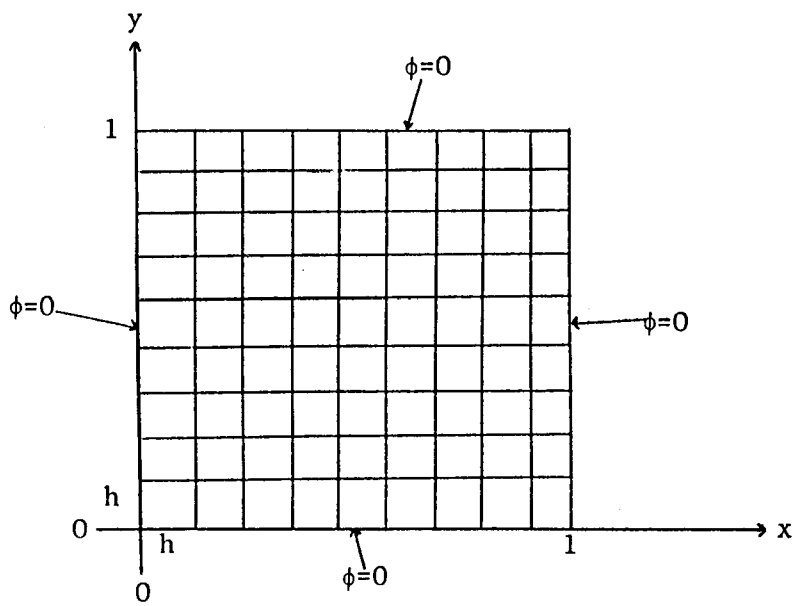


FIGURE 6.1

There are two distinct methods for the solution of systems of linear equations of this type: direct methods and iterative methods. Direct methods such as Gauss Elimination and Triangular Factorisation (see Chapter 2) yield, by a relatively complicated procedure, the exact solution to the system of equations in a finite number of steps if no rounding errors are present. Iterative methods, however, involve the repeated application of a simple formula that eventually yields the exact answer as the limit to a sequence.

The system of equations that is produced by the finite-difference method is generally large and sparse and so it is preferable to use an iterative method for its solution since they are able to take advantage of the large number of zeros in the coefficient matrix. Iterative methods are characterised by the arbitrary selection of an initial approximation $\phi^{(0)}$ to the exact solution ϕ, and the subsequent calculation of a sequence of approximations $\phi^{(1)},\phi^{(2)},\ldots$ converging to ϕ.

When applying certain iterative methods, e.g. successive over-relaxation (see section 6.3), the order in which the mesh points are updated is important and so in this chapter we shall consider various mesh point orderings that permit the parallel execution of the algorithm.

6.2 THE DERIVATION OF THE FINITE-DIFFERENCE EQUATION

Let us consider the small segment of mesh illustrated in Figure 6.2 which has a constant mesh size h. The values of ϕ at the neighbouring mesh points (x,y+h) and (x,y-h) may be expanded in terms of $\phi(x,y)$ and its derivatives by the use of Taylor's Theorem thus,

$$\phi(x,y+h) = \phi(x,y) + h\frac{\partial\phi}{\partial y}(x,y) + \frac{h^2}{2!}\frac{\partial^2\phi}{\partial y^2}(x,y) + \frac{h^3}{3!}\frac{\partial^3\phi}{\partial y^3}(x,y) + \frac{h^4}{4!}\frac{\partial^4\phi}{\partial y^4}(x,y) + \ldots\ldots \qquad (6.2.1)$$

and

$$\phi(x,y-h) = \phi(x,y) - h\frac{\partial\phi}{\partial y}(x,y) + \frac{h^2}{2!}\frac{\partial^2\phi}{\partial y^2}(x,y) - \frac{h^3}{3!}\frac{\partial^3\phi}{\partial y^3}(x,y) + \frac{h^4}{4!}\frac{\partial^4\phi}{\partial y^4}(x,y) + \ldots\ldots \qquad (6.2.2)$$

The addition of these two equations gives,

$$\phi(x,y+h) + \phi(x,y-h) = 2\phi(x,y) + h^2\frac{\partial^2\phi}{\partial y^2}(x,y) + O(h^4) \quad , \qquad (6.2.3)$$

which by discarding terms in h^4 and higher and rearranging leads to

$$\frac{\partial^2\phi}{\partial y^2}(x,y) = \frac{\phi(x,y+h)-2\phi(x,y)+\phi(x,y-h)}{h^2} \quad . \qquad (6.2.4)$$

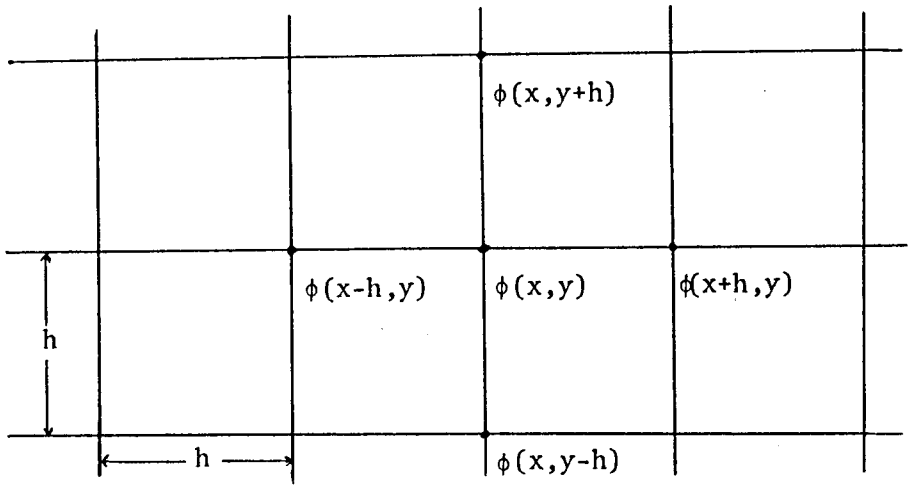


FIGURE 6.2

Similarly, by the expansion of ϕ at mesh points (x+h,y) and (x-h,y), we have,

$$\frac{\partial^2 \phi}{\partial x^2}(x,h) = \frac{\phi(x+h,y)-2\phi(x,y)+\phi(x-h,y)}{h^2} \quad . \tag{6.2.5}$$

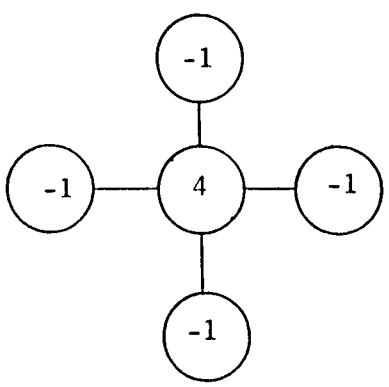
Substitution of the expressions (6.2.4) and (6.2.5) into Laplace's equation (6.1.2) yields the five point difference scheme,

$$\frac{\phi(x+h,y)+\phi(x-h,y)+\phi(x,y+h)+\phi(x,y-h)-4\phi(x,y)}{h^2} = 0$$

or

$$\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - 4\phi_{i,j} = 0 \ , \tag{6.2.6}$$

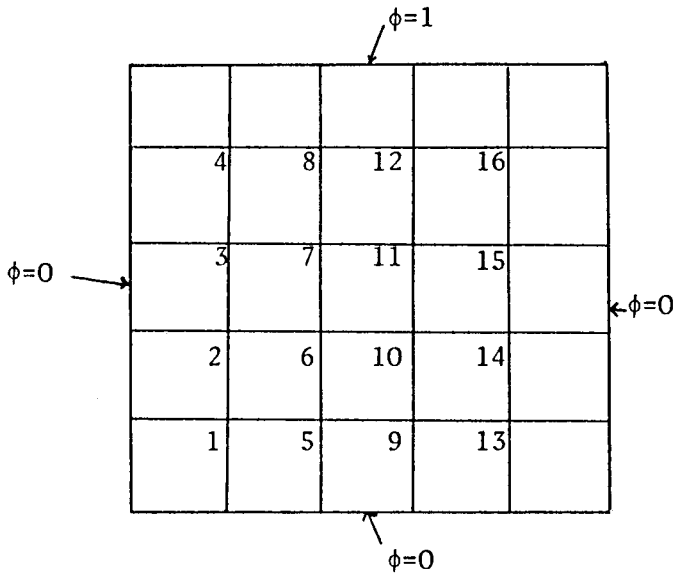
where $\phi_{i,j} = \phi(ih,jh)$, and is represented conveniently by the 'molecule' of Figure 6.3.



Five-point Difference Scheme Molecule

FIGURE 6.3

Now consider the following numbering of the internal mesh points of a (6×6) mesh with Dirichlet boundary conditions (6.1.3):-



(6.2.7)

Then, by applying equation (6.2.6) to each of the internal mesh points, we produce the following system of linear equations,

$$\begin{bmatrix} 4 & -1 & & -1 & & & & & & & & & & & & \\ -1 & 4 & -1 & & -1 & & & & & & & & & & & \\ & -1 & 4 & -1 & & -1 & & & & & & & & & & \\ & & -1 & 4 & & & -1 & & & & & & & & & \\ -1 & & & & 4 & -1 & & -1 & & & & & & & & \\ & -1 & & & -1 & 4 & -1 & & -1 & & & & & & & \\ & & -1 & & & -1 & 4 & -1 & & -1 & & & & & & \\ & & & -1 & & & -1 & 4 & & & -1 & & & & & \\ & & & & -1 & & & & 4 & -1 & & -1 & & & & \\ & & & & & -1 & & & -1 & 4 & -1 & & -1 & & & \\ & & & & & & -1 & & & -1 & 4 & -1 & & -1 & & \\ & & & & & & & -1 & & & -1 & 4 & & & -1 & \\ & & & & & & & & -1 & & & & 4 & -1 & & \\ & & & & & & & & & -1 & & & -1 & 4 & -1 & \\ & & & & & & & & & & -1 & & & -1 & 4 & -1 \\ & & & & & & & & & & & -1 & & & -1 & 4 \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \\ \phi_5 \\ \phi_6 \\ \phi_7 \\ \phi_8 \\ \phi_9 \\ \phi_{10} \\ \phi_{11} \\ \phi_{12} \\ \phi_{13} \\ \phi_{14} \\ \phi_{15} \\ \phi_{16} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad (6.2.8)$$

Thus a system of linear equations has been created by replacing the partial differential equation by a finite-difference equation at each of the internal mesh points.

6.3 THE SOLUTION OF A LARGE SPARSE SYSTEM OF LINEAR EQUATIONS

In this section we shall define some basic iterative formulae (Smith, 1965) that may be used to solve the system of equations (6.2.8).

(a) The Point Jacobi Method

Since, at each internal mesh point we have,

$$\phi_{i,j} = (\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1})/4 \ , \qquad (6.3.1)$$

then a simple iterative formula would be,

$$\phi_{i,j}^{(n+1)} = (\phi_{i+1,j}^{(n)} + \phi_{i-1,j}^{(n)} + \phi_{i,j+1}^{(n)} + \phi_{i,j-1}^{(n)})/4 \ , \qquad (6.3.2)$$

where $\phi_{i,j}^{(n)}$ represents the n^{th} iterate or approximation to ϕ at point (ih,jh). This is called the Point Jacobi method. Clearly the $(n+1)^{th}$ iterates are expressed exclusively in terms of n^{th} iterates and so the order in which they are evaluated with respect to the mesh points does not effect their values or the rate of convergence to the solution. Hence this method is called the Simultaneous Displacement Method. Unfortunately, the rate of convergence of this method is slow and hence it is rarely used.

(b) The Gauss-Seidel Method

The Point Jacobi formula (6.3.2) may be improved by using the latest values of $\phi_{i,j}$ as soon as they are available. If we assume that the $(n+1)^{th}$ iterative values have been calculated along columns $1,2,\ldots,(j-1)$ and as far as point $(i-1,j)$ along column j, and that the $(n+1)^{th}$ value at point (i,j) is the next to be calculated, then the Gauss-Seidel formula gives

$$\phi_{i,j}^{(n+1)} = (\phi_{i+1,j}^{(n)} + \phi_{i-1,j}^{(n+1)} + \phi_{i,j+1}^{(n)} + \phi_{i,j-1}^{(n+1)})/4 \ . \qquad (6.3.3)$$

With this method, we have the added advantage of only needing to store the latest value of each $\phi_{i,j}$. This method is a Successive Displacement Method.

(c) The Successive Over-Relaxation Method (S.O.R.)

If $\phi_{i,j}^{(n)}$ is added and subtracted to the right hand side of equation (6.3.3) we have,

$$\phi_{i,j}^{(n+1)} = \phi_{i,j}^{(n)} + (\phi_{i+1,j}^{(n)} + \phi_{i-1,j}^{(n+1)} + \phi_{i,j+1}^{(n)} + \phi_{i,j-1}^{(n+1)} - 4\phi_{i,j}^{(n)})/4$$

$$= \phi_{i,j}^{(n)} + r_{i,j} \quad . \tag{6.3.4}$$

Obviously, $r_{i,j}$ is the change in value of $\phi_{i,j}$ for one Gauss-Seidel iteration. The rate of convergence of the Gauss-Seidel method can be 'accelerated' by making a larger change to $\phi_{i,j}$ thus,

$$\phi_{i,j}^{(n+1)} = \phi_{i,j}^{(n)} + \omega r_{i,j} \quad , \tag{6.3.5}$$

where ω is positive constant called the acceleration factor which in practice lies between 1 and 2. This equation is called the Successive Over-Relaxation formula and may be rewritten in the form,

$$\phi_{i,j}^{(n+1)} = (1-\omega)\phi_{i,j}^{(n)} + \omega(\phi_{i+1,j}^{(n)} + \phi_{i-1,j}^{(n+1)} + \phi_{i,j+1}^{(n)} + \phi_{i,j-1}^{(n+1)})/4, \tag{6.3.6}$$

from which it is clear that it is a linear combination of the Gauss-Seidel iterate (6.3.3) and the n^{th} iterate. (Note that when $\omega=1$, the S.O.R. method becomes the Gauss-Seidel method). In this method it is also only necessary to store the latest values of $\phi_{i,j}$ and it is a Successive Displacement Method.

In order to find the conditions necessary for convergence of these methods, consider their matrix form. Assume that we wish to solve the system of equations

$$A\underline{\phi} = \underline{b} \quad , \tag{6.3.7}$$

where A is an (m×m) matrix and $\underline{\phi}$ and $\underline{b}$ are (m×1) vectors. Then (6.3.7) can be expressed in the form,

$$(I-L-U)\underline{\phi} = \underline{b} \quad , \tag{6.3.8}$$

where I is the (n×n) unit matrix and -L and -U are strictly lower

and upper triangular matrices respectively of the form,

$$-L = \begin{bmatrix} 0 & & & \\ a_{21} & 0 & & 0 \\ a_{31} & a_{32} & 0 & \\ \vdots & \vdots & \ddots & \ddots \\ a_{m,1} & \cdots & a_{m,m-1} & 0 \end{bmatrix} \quad \text{and} \quad -U = \begin{bmatrix} 0 & a_{12} & a_{13} & \cdots & a_{1,m} \\ & 0 & a_{23} & & \\ & & 0 & \ddots & \vdots \\ & & & \ddots & a_{m-1,m} \\ 0 & & & & 0 \end{bmatrix}$$

Hence, the matrix form of the Point Jacobi method is

$$\underline{\phi}^{(n+1)} = (L+U)\underline{\phi}^{(n)} + \underline{b} \quad , \tag{6.3.9}$$

of the Gauss-Seidel method it is,

$$\underline{\phi}^{(n+1)} = L\underline{\phi}^{(n+1)} + U\underline{\phi}^{(n)} + \underline{b}$$

which on rearrangement gives,

$$\underline{\phi}^{(n+1)} = (I-L)^{-1}U\underline{\phi}^{(n)} + (I-L)^{-1}\underline{b} \quad , \tag{6.3.10}$$

and of the S.O.R. method it is,

$$\underline{\phi}^{(n+1)} = \underline{\phi}^{(n)} + \omega(L\underline{\phi}^{(n+1)} + U\underline{\phi}^{(n)} + \underline{b} - \underline{\phi}^{(n)})$$

which leads to,

$$\underline{\phi}^{(n+1)} = (I-\omega L)^{-1}(\omega U-(\omega-1)I)\underline{\phi}^{(n)} + (I-\omega L)^{-1}\omega\underline{b} \quad . \tag{6.3.11}$$

Now if the error at any stage is the difference between the true and approximate solutions, i.e.,

$$\underline{e}^{(n)} = \underline{\phi} - \underline{\phi}^{(n)} \quad , \tag{6.3.12}$$

then by subtracting (6.3.9) from (6.3.8), we have the error vector by the Point Jacobi method as,

$$\underline{e}^{(n+1)} = (L+U)\underline{e}^{(n)} = (L+U)(L+U)\underline{e}^{(n-1)} = \ldots = (L+U)^{n+1}\underline{e}^{(0)} , \tag{6.3.13}$$

and similarly, for the Gauss-Seidel method we have,

$$\underline{e}^{(n+1)} = [(I-L)^{-1}U]^{n+1}\underline{e}^{(0)} \quad . \tag{6.3.14}$$

In the case of S.O.R., the true solution satisfies

$$(I-\omega L)\underline{\phi} = (\omega U-(\omega-1)I)\underline{\phi} + \omega\underline{b}$$

and so the error vector for S.O.R. is

$$\underline{e}^{(n+1)} = [(I-\omega L)^{-1}(\omega U-(\omega-1)I)]^{n+1}\underline{e}^{(0)} \quad . \tag{6.3.15}$$

Thus, for each of these methods, the relationship between successive error vectors is

$$\underline{e}^{(n+1)} = H\underline{e}^{(n)} \quad , \tag{6.3.16}$$

where H, the iteration matrix is defined as,

$$H = \begin{cases} (L+U), & \text{for the Point Jacobi method} \\ (I-L)^{-1}U, & \text{for the Gauss-Seidel method} \\ [(I-\omega L)^{-1}(\omega U-(\omega-1)I)], & \text{for the S.O.R. method.} \end{cases}$$

Assuming that the m eigenvalues λ_s (s=1,2,...,m) of H are all different, then the corresponding m eigenvectors $\underline{v}_s$ form a linearly independent set of vectors, where by the definition of an eigenvalue,

$$H\underline{v}_s = \lambda_s \underline{v}_s \quad . \tag{6.3.17}$$

So $\underline{e}^{(0)}$ may be expressed as a linear combination of the eigenvectors of H thus,

$$\underline{e}^{(0)} = \sum_{s=1}^{m} c_s \underline{v}_s \quad , \tag{6.3.18}$$

therefore,

$$\underline{e}^{(1)} = H\underline{e}^{(0)} = \sum_{s=1}^{m} c_s H\underline{v}_s = \sum_{s=1}^{m} c_s \lambda_s \underline{v}_s \quad ,$$

and hence, for $\underline{e}^{(n)}$ we have,

$$\underline{e}^{(n)} = \sum_{s=1}^{m} c_s \lambda_s^n \underline{v}_s \quad . \tag{6.3.19}$$

Clearly, for convergence of the iterative formula, we require $\underline{e}^{(n)}$ to tend to zero, which means that the eigenvalue λ_s of H with largest absolute value, called the spectral radius of H, must be less than unity.

6.4 THE ESTIMATION OF THE OPTIMUM VALUE OF ω FOR S.O.R.

The rate at which the S.O.R. method converges is dependent on the value of the acceleration factor ω and so to maximise the convergence rate, the best value of ω, say ω_b, must be estimated. The basis for the theoretical estimation of ω_b rests on work done by

Young [1954], who developed the theory of matrices possessing the following property, which he termed 'property A'.

For the system of equations (6.3.7), matrix A is said to possess property A if there exists two disjoint subsets S and T of W (the first n integers) such that S+T=W and if $a_{i,j} \neq 0$, then either i=j or $i \varepsilon S$ and $j \varepsilon T$ or $i \varepsilon T$ and $j \varepsilon S$.

In addition to property A, it is necessary that the order in which the $(n+1)^{th}$ iterative values are evaluated satisfies a certain condition called consistent ordering, which is defined thus.

If matrix A has property A then it is always possible to reorder the equations and unknowns so that the new coefficient matrix has either the tridiagonal form,

$$\begin{bmatrix} D_1 & F_1 & & & & \\ E_1 & D_2 & F_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & E_{\ell-2} & D_{\ell-1} & F_{\ell-1} & \\ & & & E_{\ell-1} & D_{\ell} \end{bmatrix} , \qquad (6.4.1)$$

or the partitioned form,

$$\begin{bmatrix} D_1 & F \\ E & D_2 \end{bmatrix} , \qquad (6.4.2)$$

where the D's are square diagonal submatrices, not necessarily of the same order, and the E's and F's are rectangular submatrices. Assuming that the equations have been ordered so as to give a matrix of the form (6.4.1) or (6.4.2), then a different ordering of the equations is said to be consistent with form (6.4.1) or (6.4.2) when the $(n+1)^{th}$ iterative values for the two orderings are identical for n=0,1,2,..., initial inputs being the same of course.

The importance of these two properties is that, if matrix A is consistently ordered and has property A, then the eigenvalues, λ, of

the S.O.R. iteration matrix and μ of the Point Jacobi matrix (see (6.3.16)) are related by the equation,

$$(\lambda+\omega-1)^2 = \lambda\omega^2\mu^2, \quad \text{if} \quad |\mu|<1 \ . \tag{6.4.3}$$

i.e.
$$\lambda^{\frac{1}{2}} = \frac{\omega\mu\pm\sqrt{\omega^2\mu^2-4(\omega-1)}}{4} \ . \tag{6.4.4}$$

Now, from equation (6.3.19) we see that the convergence rate is dependent on λ and so to optimise the rate of convergence, $\bar{\lambda}$, the eigenvalue of maximum modulus of the S.O.R. iteration matrix, must be minimised. This is achieved by making the square root in equation (6.4.4) equal to zero for $\bar{\mu}$, the eigenvalue of maximum modulus of the Point Jacobi iteration matrix, i.e.,

$$\omega^2\bar{\mu}^2 = 4(\omega-1) \ , \tag{6.4.5}$$

which yields the result,

$$\omega_b = \frac{2}{1+\sqrt{1-\bar{\mu}^2}} \ . \tag{6.4.6}$$

If this equation is used to eliminate $\bar{\mu}^2$ from equation (6.4.3), we obtain the result,

$$\bar{\lambda} = \omega_b-1 \ . \tag{6.4.7}$$

The eigenvalue of maximum modulus value of H is called the spectral radius of H. Now, since the Gauss-Seidel method is the same as S.O.R. with ω=1, substitution of this value for ω into (6.4.3) gives the result,

$$\rho(G) = \rho(J)^2 \ , \tag{6.4.8}$$

where $\rho(G)$ and $\rho(J)$ are the spectral radii of the Gauss-Seidel and Point Jacobi iteration matrices respectively. Thus, ω_b may be expressed in terms of $\rho(G)$ thus,

$$\omega_b = \frac{2}{1+\sqrt{1-\rho(G)}} \ . \tag{6.4.9}$$

Using equation (6.3.19) it is not difficult to show that the successive errors at any mesh point, after a large number of iterations,

are related by the equation,

$$|e^{(n+1)}| = \rho|e^{(n)}| \quad , \tag{6.4.10}$$

where ρ is the spectral radius of H. Therefore the common logarithm (base 10) of ρ is an indication of the number of decimal digits by which the error is eventually decreased by each iteration. For theoretical purposes the asymptotic rate of convergence, R, is defined as,

$$R = -\log_e(\rho) \quad . \tag{6.4.11}$$

It is not difficult to see that R for the Gauss-Seidel method is twice that of the Point Jacobi method. Furthermore, by considering $\log_e(\omega_b-1)$, it can be shown that R for the S.O.R. method is approximately $2/\varepsilon$ times that of the Gauss-Seidel method, where $\mu^2=1-\varepsilon^2$, ε being small for large n.

Another useful result that we may obtain is an estimation of the number of iterations, n, necessary to make $e^{(n)}<\varepsilon$, where ε is the required accuracy. From equation (6.4.10) it is not difficult to show that

$$n \simeq \frac{\log\varepsilon}{\log(\omega-1)} \quad . \tag{6.4.12}$$

The estimation of ω_b and the other quantities defined here clearly depend on whether $\rho(J)$ or $\rho(G)$ can be estimated. Several methods have been suggested by Carré [1961] and Varga [1962], one of which is the Power Method that may be described as follows.

Assuming the matrix of the finite difference equations is consistently ordered and has property A, calculate the sequence of approximations $\underline{\phi}^{(1)},\underline{\phi}^{(2)},\ldots\underline{\phi}^{(i)}$ to the solution of the system of equations $A\underline{\phi}=\underline{b}$ by the Gauss-Seidel method and then we have

$$\rho(G) = \lim_{i\to\infty} \frac{||d^{(i)}||}{||d^{(i-1)}||} \quad , \tag{6.4.13}$$

where $\underline{d}^{(i)}$ is defined as $\underline{d}^{(i)}=\underline{\phi}^{(i)}-\underline{\phi}^{(i-1)}$ and

$$||d^{(i)}|| = [\sum_{j=1}^{n} (\phi_j^{(i)} - \phi_j^{(i-1)})^2]^{\frac{1}{2}} \quad . \tag{6.4.14}$$

Thus, using the power method we can approximate $\rho(G)$, which, in turn, can be substituted into equation (6.4.9) to give an estimate of ω_b, the optimum acceleration factor.

6.5 THE SOLUTION OF THE DIRICHLET PROBLEM BY S.O.R. ON A PARALLEL COMPUTER

It is obviously a trivial problem to perform the Point Jacobi method on a parallel computer, since each iteration comprises of m^2 independent evaluations defined by (6.3.2). The use of Successive Displacement methods on a parallel computer is not so simple since the order in which the $(n+1)^{th}$ iterates are evaluated, particularly with S.O.R., is important. We have seen, in the previous section, that, with the S.O.R. method, it is desirable for matrix A, defined in (6.3.7), to possess property A and be consistently ordered, and so, when S.O.R. is performed on a parallel computer, it is useful but not vital to preserve these properties.

If the order in which the $(n+1)^{th}$ iterates are evaluated is called an ordering, then an ordering that produces a coefficient matrix A which is consistently ordered is a consistent ordering. The ordering of the mesh points in (6.2.7) is a consistent ordering. This ordering, however, is of little use for a parallel computer because it is essentially sequential. A much more useful ordering is the red-black ordering defined as,

10	④	14	⑧	
②	12	⑥	16	
9	③	13	⑦	
①	11	⑤	15	

(6.5.1)

Clearly, the red-black ordering consists of two passes over the meshes. During the first pass we evaluate the $(n+1)^{th}$ iterates at alternate mesh points (circled in (6.5.1)) beginning at ①, and in the second pass the remaining points (uncircled in (6.5.1)) are dealt with. If the finite difference equation (6.2.6) is applied in this order, then using S.O.R. we have for the first pass,

$$\phi_{i,j}^{(n+1)} = (1-\omega)\phi_{i,j}^{(n)} + \omega(\phi_{i+1,j}^{(n)} + \phi_{i-1,j}^{(n)} + \phi_{i,j+1}^{(n)} + \phi_{i,j-1}^{(n)})/4, \tag{6.5.2}$$

and during the second pass,

$$\phi_{i,j}^{(n+1)} = (1-\omega)\phi_{i,j}^{(n)} + \omega(\phi_{i+1,j}^{(n+1)} + \phi_{i-1,j}^{(n+1)} + \phi_{i,j+1}^{(n+1)} + \phi_{i,j-1}^{(n+1)})/4. \tag{6.5.3}$$

Thus the first pass consists of independent evaluations which may be carried out simultaneously and similarly so does the second pass. It is unimportant how the evaluations in each pass are shared between the processors provided it is done evenly and that the first pass is completed before the second pass is commenced (this is to ensure that during the second pass, the $(n+1)^{th}$ iterates are available when required).

An alternative application of the red-black ordering for a two processor computer can be produced by applying the technique of folding described in chapter 3, which we shall call the folding point ordering. In this ordering the two passes of the red-black ordering are executed simultaneously, in the following manner,

⑦	4	③	8	
2	⑤	6	①	
⑧	3	④	7	
1	⑥	5	②	

(6.5.4)

Processor 1 evaluates the $(n+1)^{th}$ iterates at the uncircled mesh points while processor 2 evaluates the $(n+1)^{th}$ iterates at the circled mesh points in the order 1 and ① , 2 and ② etc. The $(n+1)^{th}$ iterates are defined by equation (6.5.2) before the processors cross, and by (6.5.3) after they have crossed, for both processors.

For both the red-black and folding point orderings the number of parallel operations per iteration is

$$3\left\lfloor\frac{m^2}{2}\right\rfloor \text{ multiplications} + 5\left\lfloor\frac{m^2}{2}\right\rfloor \text{ additions} , \qquad (6.5.5)$$

where m^2 is the number of internal mesh points.

In order to compare the two mesh point orderings, they are both used to solve the Dirichlet problem for different mesh sizes. In each case, $\rho(G)$ is estimated using the power method and, by substituting this value into equation (6.4.9), ω_b is obtained. An experimental optimum value of ω is also found by solving the problem using different values of ω. The value of ω is initially set to 1 and incremented by $\Delta\omega$ until the number of iterations required to satisfy the conditions of convergence begins to increase. Then, in the vicinity of the value of ω that requires the least number of iterations, a smaller value of $\Delta\omega$ is used. The process is repeated until the region, in which the least number of iterations are required for convergence, is found to the required degree of accuracy. The experimental best value of ω, say ω_e, is the average value of ω for which the least number of iterations is required.

The condition for convergence of the S.O.R. method is,

$$|\phi_{i,j}^{(n+1)} - \phi_{i,j}^{(n)}| < \varepsilon \quad , \text{ for all } i,j , \qquad (6.5.6)$$

where $\varepsilon=5\times10^{-5}$, and also for the power method, the difference between successive estimates of $\rho(G)$ is chosen to be less than the same value of ε. Four different mesh sizes are used which produce

(10×10), (20×20), (40×40) and (60×60) networks. The results obtained from these experiments are recorded in Table (6.1), where n_A is the minimum number of iterations required for convergence, n_E is the number of iterations estimated from equation (6.4.12), and ω_e and ω_b are as previously defined. Obviously there is, as might be expected, little difference between the results achieved by the two orderings.

Method	Mesh size	n_A	n_E	ω_e	ω_b
Red-black point SOR	10	15	14	1.495	1.490
	20	29	30	1.717	1.717
	40	57	58	1.846	1.842
	60	82	76	1.890	1.877
Folding point SOR	10	15	14	1.503	1.490
	20	32	30	1.732	1.718
	40	58	58	1.848	1.842
	60	83	76	1.894	1.877

TABLE 6.1

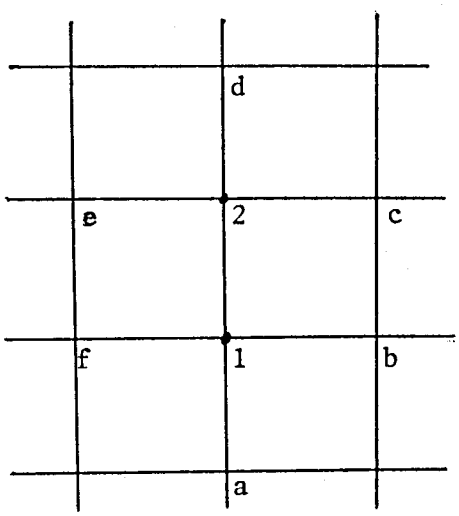
One fact that is not so obvious, however, is that the folding point ordering is not consistent. This is the result of the simultaneous evaluation of adjacent mesh values, since the $(n+1)^{th}$ iterates at the two points are evaluated using the n^{th} iterative values of each other, whereas when evaluated sequentially, the second point to be evaluated would use the $(n+1)^{th}$ iterative value of the first point. Hence the sequential ordering is not preserved. Clearly, in this case, the fact that the ordering is inconsistent, has no serious effect on the performance of the algorithm. One question that cannot be answered however until the algorithm is actually implemented, is, 'will the 2 processors cross at different points during each iteration?' and if so 'will it have a more serious effect on the algorithm's performance?'

The necessity of consistent ordering is an important question and from some of the following orderings it will be seen that the lack of consistency can be a serious problem. The main disadvantage is that the theory of section 6.4 does not hold, which can make it impossible to estimate ω_b accurately.

6.6 BLOCK AND LINE ITERATIVE SCHEMES

The number of iterations required for the convergence of the iterative process may be reduced by evaluating iterates at groups of mesh points by a direct method. This technique leads to block and line iterative methods to which the results obtained in section 6.4 also apply.

Consider for instance, the following group of mesh points,



(6.6.1)

If equation (6.2.6) is applied to points 1 and 2 we obtain the formulae,

$$\left.\begin{aligned} 4\phi_1 &= \phi_a + \phi_b + \phi_2 + \phi_f \\ \text{and} \quad 4\phi_2 &= \phi_1 + \phi_c + \phi_d + \phi_e \end{aligned}\right\} \tag{6.6.2}$$

which may be rearranged to give,

$$\left.\begin{aligned} \phi_1 &= (4(\phi_a + \phi_b + \phi_f) + \phi_c + \phi_d + \phi_e)/15 \\ \text{and} \quad \phi_2 &= (4(\phi_c + \phi_d + \phi_e) + \phi_a + \phi_b + \phi_f)/15 \end{aligned}\right\}, \tag{6.6.3}$$

or in terms of i and j,

$$\left.\begin{aligned} \phi_{i,j} &= (4(\phi_{i-1,j}+\phi_{i,j+1}+\phi_{i,j-1}) + \phi_{i+1,j+1}+\phi_{i+2,j}+\phi_{i+1,j-1})/15 \\ \text{and} \quad \phi_{i+1,j} &= (4(\phi_{i+1,j+1}+\phi_{i+2,j}+\phi_{i+1,j-1})+\phi_{i-1,j}+\phi_{i,j+1}+\phi_{i,j-1})/15 \end{aligned}\right\} \quad (6.6.4)$$

Clearly, these two equations are independent of each other and so may be evaluated simultaneously. Thus, by partitioning the system of meshes into (2×1) blocks we can evaluate the iterates at the points within each block simultaneously using two processors. The order in which the blocks are considered is important and so, remembering that the processors of an MIMD computer are not synchronous, we shall use the red-black ordering as in (6.5.1), except that each point represents a (2×1) block. Any consistent ordering of the blocks may, of course, be used but with the ordering defined in (6.2.7) for instance, we cannot be sure that all of the latest iterative values will be available when required.

So, using a red-black ordering of the blocks, the $(n+1)^{th}$ iterates of the SOR iterative scheme will be defined by,

$$\left.\begin{aligned} \phi_{i,j}^{(n+1)} &= \phi_{i,j}^{(n)} + \frac{\omega}{15}(4r_{i,j}^{(n)} + r_{i+1,j}^{(n)} - 15\phi_{i,j}^{(n)}) \\ \text{and} \quad \phi_{i+1,j}^{(n+1)} &= \phi_{i+1,j}^{(n)} + \frac{\omega}{15}(4r_{i+1,j}^{(n)} + r_{i,j}^{(n)} - 15\phi_{i+1,j}^{(n)}) \end{aligned}\right\} \quad (6.6.5)$$

where $\quad r_{i,j}^{(n)} = (\phi_{i-1,j}^{(n)} + \phi_{i,j+1}^{(n)} + \phi_{i,j-1}^{(n)})$

and $\quad r_{i+1,j}^{(n)} = (\phi_{i+1,j+1}^{(n)} + \phi_{i+2,j}^{(n)} + \phi_{i+1,j-1}^{(n)})$,

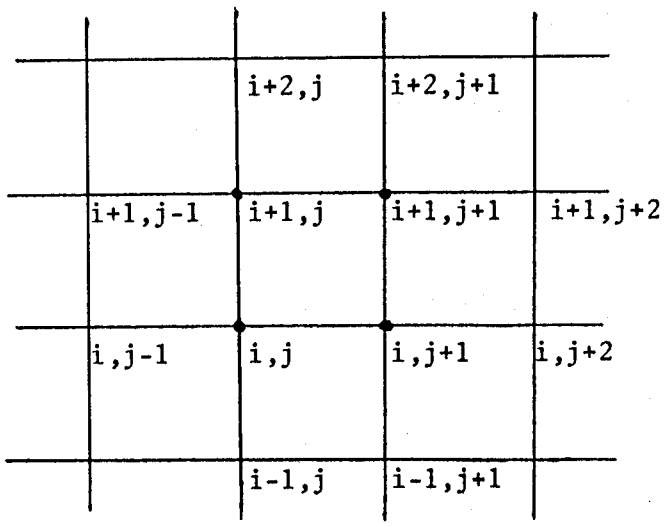
during the first pass and,

$$\begin{aligned} \phi_{i,j}^{(n+1)} &= \phi_{i,j}^{(n)} + \frac{\omega}{15}(4r_{i,j}^{(n+1)} + r_{i+1,j}^{(n+1)} - 15\phi_{i,j}^{(n)}) \\ \text{and} \quad \phi_{i+1,j}^{(n+1)} &= \phi_{i+1,j}^{(n)} + \frac{\omega}{15}(4r_{i+1,j}^{(n+1)} + r_{i,j}^{(n+1)} - 15\phi_{i+1,j}^{(n)}) \end{aligned} \quad (6.6.6)$$

during the second pass.

In order to use more than two processors, we can either solve for more than one block at a time, or, alternatively, partition the system

of meshes into larger blocks; so consider the following (2×2) block of mesh points,



(6.6.7)

Applying the same technique as was applied to the (2×1) block, we obtain the formulae,

$$\left.\begin{aligned}
\phi_{i,j} &= (2(\phi_{i-1,j+1}+\phi_{i,j+2}+\phi_{i+2,j}+\phi_{i+1,j-1})+7(\phi_{i-1,j}+\phi_{i,j-1}) \\
&\qquad +\phi_{i+1,j+2}+\phi_{i+2,j+1})/24 \\
&= r_{i,j}/24 \quad , \\
\phi_{i,j+1} &= (2(\phi_{i-1,j}+\phi_{i+1,j+2}+\phi_{i+2,j+1}+\phi_{i,j-1})+7(\phi_{i-1,j+1}+\phi_{i,j+2}) \\
&\qquad +\phi_{i+2,j}+\phi_{i+1,j-1})/24 \\
&= r_{i,j+1}/24 \quad , \\
\phi_{i+1,j+1} &= (2(\phi_{i-1,j+1}+\phi_{i,j+2}+\phi_{i+2,j}+\phi_{i+1,j-1})+7(\phi_{i+1,j+2}+\phi_{i+2,j+1}) \\
&\qquad +\phi_{i-1,j}+\phi_{i,j-1})/24 \\
&= r_{i+1,j+1}/24 \quad , \\
\text{and} \quad \phi_{i+1,j} &= (2(\phi_{i-1,j}+\phi_{i+1,j+2}+\phi_{i+2,j+1}+\phi_{i,j-1})+7(\phi_{i+2,j}+\phi_{i+1,j-1}) \\
&\qquad + \phi_{i-1,j+1} + \phi_{i,j+2})/24 \\
&= r_{i+1,j}/24
\end{aligned}\right\} \qquad (6.6.8)$$

Again, using a red-black ordering of the blocks, the $(n+1)^{th}$ iterates during the first pass are defined by,

$$\left.\begin{aligned}
&\phi_{i,j}^{(n+1)} = \phi_{i,j}^{(n)} + \frac{\omega}{24}(r_{i,j}^{(n)} - 24\phi_{i,j}^{(n)}) \ , \\
&\phi_{i,j+1}^{(n+1)} = \phi_{i,j+1}^{(n)} + \frac{\omega}{24}(r_{i,j+1}^{(n)} - 24\phi_{i,j+1}^{(n)}) \ , \\
&\phi_{i+1,j+1}^{(n+1)} = \phi_{i+1,j+1}^{(n)} + \frac{\omega}{24}(r_{i+1,j+1}^{(n)} - 24\phi_{i+1,j+1}^{(n)}) \ , \\
\text{and} \quad &\phi_{i+1,j}^{(n+1)} = \phi_{i+1,j}^{(n)} + \frac{\omega}{24}(r_{i+1,j}^{(n)} - 24\phi_{i+1,j}^{(n)}) \ ,
\end{aligned}\right\} \tag{6.6.9}$$

and during the second pass by,

$$\left.\begin{aligned}
&\phi_{i,j}^{(n+1)} = \phi_{i,j}^{(n)} + \frac{\omega}{24}(r_{i,j}^{(n+1)} - 24\phi_{i,j}^{(n)}) \ , \\
&\phi_{i,j+1}^{(n+1)} = \phi_{i,j+1}^{(n)} + \frac{\omega}{24}(r_{i,j}^{(n+1)} - 24\phi_{i,j+1}^{(n)}) \ , \\
&\phi_{i+1,j+1}^{(n+1)} = \phi_{i+1,j+1}^{(n)} + \frac{\omega}{24}(r_{i+1,j+1}^{(n+1)} - 24\phi_{i+1,j+1}^{(n)}) \ , \\
\text{and} \quad &\phi_{i+1,j}^{(n+1)} = \phi_{i+1,j}^{(n)} + \frac{\omega}{24}(r_{i+1,j}^{(n+1)} - 24\ \phi_{i+1,j}^{(n)}) \ .
\end{aligned}\right\}$$

Obviously, with this scheme, we may use 4 processors simultaneously. If there are m^2 internal mesh points, m must be even. The latter point also applies to the (2×1) block scheme.

Considering now the number of parallel operations per iteration, we have for the (2×1) block scheme,

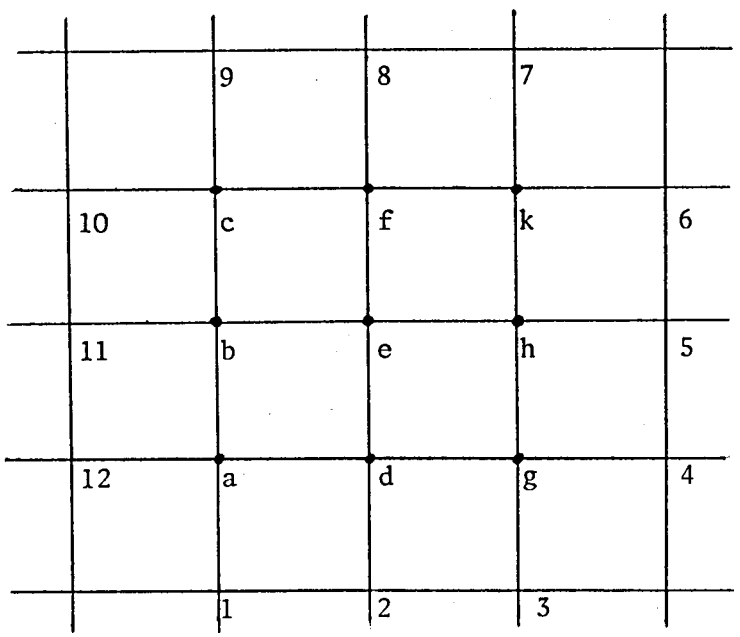
$$(2m^2 \text{ multiplications} + \tfrac{7}{2}m^2 \text{ additions}) \ , \tag{6.6.10}$$

when using 2 processors and for the (2×2) block scheme,

$$(\tfrac{5}{4}m^2 \text{ multiplications} + \tfrac{9}{4}m^2 \text{ additions}) \ , \tag{6.6.11}$$

when using 4 processors.

The next size of block to be considered is the (3×3) block,



(6.6.12)

Applying the same technique as before, we obtain the formulae,

$$\left.\begin{aligned}
\phi_a &= (67(\phi_1+\phi_{12})+22(\phi_2+\phi_{11})+7(\phi_3+\phi_4+\phi_9+\phi_{10})+6(\phi_5+\phi_8) \\
&\qquad +3(\phi_6+\phi_7))/224 \ , \\
\phi_b &= (37\phi_{11}+11(\phi_1+\phi_9+\phi_{10}+\phi_{12})+7(\phi_2+\phi_8)+5\phi_5 \\
&\qquad +3(\phi_3+\phi_4+\phi_6+\phi_7)/112 \ , \\
\phi_c &= (67(\phi_9+\phi_{10})+22(\phi_8+\phi_{11})+7(\phi_1+\phi_6+\phi_7+\phi_{12})+6(\phi_2+\phi_5) \\
&\qquad +3(\phi_3+\phi_4))/224 \ , \\
\phi_d &= (37\phi_2+11(\phi_1+\phi_3+\phi_4+\phi_{12})+7(\phi_5+\phi_{11})+5\phi_8 \\
&\qquad +3(\phi_6+\phi_7+\phi_9+\phi_{10}))/112 \ , \\
\phi_e &= (2(\phi_2+\phi_5+\phi_8+\phi_{11})+\phi_1+\phi_3+\phi_4+\phi_6+\phi_7+\phi_9+\phi_{10}+\phi_{12})/16 \\
\phi_f &= (37\phi_8+11(\phi_6+\phi_7+\phi_9+\phi_{10})+7(\phi_5+\phi_{11})+5\phi_2 \\
&\qquad +3(\phi_1+\phi_3+\phi_4+\phi_{12}))/112 \ , \\
\phi_g &= (67(\phi_3+\phi_4)+22(\phi_2+\phi_5)+7(\phi_1+\phi_6+\phi_7+\phi_{12})+6(\phi_8+\phi_{11}) \\
&\qquad +3(\phi_9+\phi_{10}))/224 \ , \\
\phi_h &= (37\phi_5+11(\phi_3+\phi_4+\phi_6+\phi_7)+7(\phi_2+\phi_8)+5\phi_{11} \\
&\qquad +3(\phi_1+\phi_9+\phi_{10}+\phi_{12}))/112, \\
\text{and} \quad \phi_k &= (67(\phi_6+\phi_7)+22(\phi_5+\phi_8)+7(\phi_3+\phi_4+\phi_9+\phi_{10})+6(\phi_2+\phi_{11}) \\
&\qquad +3(\phi_1+\phi_{12}))/224 \ ,
\end{aligned}\right\} \quad (6.6.13)$$

from which it is not difficult to produce the corresponding S.O.R. formulae. Thus, by evaluating the iterative values at all of the mesh points within a block simultaneously, we can use 9 processors. Again, it is preferable to employ a red-black ordering of the blocks, and for this scheme, m must be devisable by 3.

An unfortunate property of this block size is that the equations (6.6.13) are not all of the same form and so the rates at which each of the processors traverse the system of meshes will not be the same. However, by considering the processor that has the most work to do, the number of parallel operations will be,

$(\frac{8}{9}m^2$ multiplications + $\frac{13}{9}$ additions) per iteration. (6.6.14)

If we compare the number of parallel operations per iteration of the three block S.O.R. schemes considered so far, it can be seen that, as might be expected, this quantity decreases as the block size (and therefore the number of processors) is increased. However, going from the (2×1) block to the (2×2) block, for instance, does not halve the number of operations and so it is important to compare the respective rates of convergence. For this purpose the experiments performed using the point iterative schemes were repeated using the block schemes, the results of which are contained in Table 6.2. The headings of Table 6.2 are the same as those of Table 6.1. The differences in the mesh sizes for the (3×3) block scheme are to allow for the fact that m (the square root of the total number of internal mesh points) must be divisible by 3.

Method	Mesh size	n_A	n_E	ω_e	ω_b
(2×1) Block SOR	10	13	13	1.449	1.439
	20	26	26	1.681	1.681
	40	50	52	1.822	1.824
	60	73	70	1.874	1.867
(2×2) Block SOR	10	11	10	1.371	1.365
	20	22	22	1.617	1.625
	40	42	43	1.784	1.793
	60	61	61	1.846	1.850
(3×3) Block SOR	11	10	10	1.345	1.333
	20	18	18	1.561	1.563
	41	36	37	1.749	1.760
	62	52	54	1.826	1.830

TABLE 6.2

As expected, by increasing the block size, the number of iterations required for convergence of the iterative schemes is decreased. The results contained in Table 6.2 can be combined with the number of operations

per iteration required by each method given in (6.6.10), (6.6.11) and (6.6.14), to give the total number of parallel operations required by each of the block S.O.R. methods, and are recorded in Table 6.3.

Mesh size		10		20		40		60	
Method	P	M	A	M	A	M	A	M	A
(2×1) Block SOR	2	$26m^2$	$\frac{91}{2}m^2$	$52m^2$	$91m^2$	$100m^2$	$175m^2$	$146m^2$	$\frac{511}{2}m^2$
(2×2) Block SOR	4	$\frac{55}{4}m^2$	$\frac{99}{4}m^2$	$\frac{55}{2}m^2$	$\frac{99}{2}m^2$	$\frac{105}{2}m^2$	$\frac{189}{2}m^2$	$\frac{305}{4}m^2$	$\frac{549}{4}m^2$
(3×3) Block* SOR	9	$\frac{80}{9}m^2$	$\frac{130}{9}m^2$	$16m^2$	$26m^2$	$32m^2$	$52m^2$	$\frac{416}{9}m^2$	$\frac{676}{9}m^2$

where P=no. of processors, M=multiplications and A=additions

**mesh sizes are 11, 20, 41 and 62 as in Table 6.2*

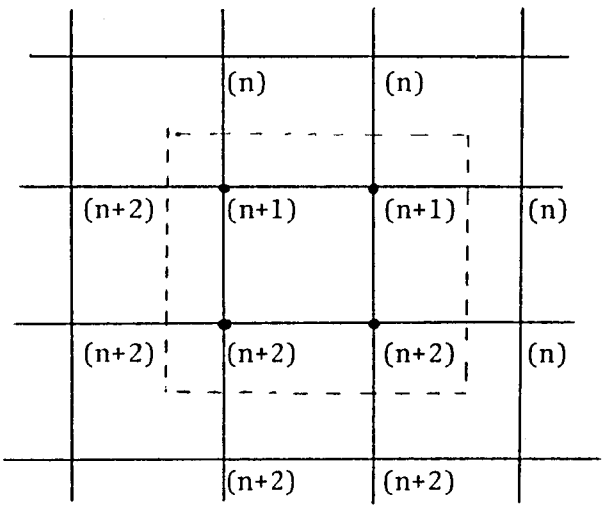
TABLE 6.3

Clearly, by increasing the block size from (2×1) to (2×2), we see that the number of parallel operations is approximately halved and so would be justified if sufficient processors are available. The effect of increasing the block size to (3×3) is not quite so successful but still impressive. However, it must be remembered that the equations generated by the (3×3) block are not identical in form and also the parallel overheads will be considerably more for 9 processors than for 2 or 4 processors.

An interesting strategy that should further improve the power of these block SOR methods would be to overlap the blocks. Consider, for instance, the (2×2) block method defined in the equations (6.6.9). In order to achieve a block ordering similar to (6.2.7), the i and j indices are incremented by 2 from 1 to (m-1); the i index being incremented first. If the i index is incremented by 1 instead of 2, the blocks in each column will overlap such that the elements (i,j) and (i,j+1) of each block will be

the elements (i+1,j) and (i+1,j+1) of the previous block (see (6.6.7)).

Obviously, during an iteration, each element will be evaluated twice using the latest iterates at neighbouring mesh points which are illustrated as follows:



(6.6.15)

where bracketed values represent the iterative level at each mesh point. This method is called the (2×2) Overlapping Block SOR method and by applying a similar technique to the (2×1) Block SOR method we obtain the (2×1) Overlapping Block Method. Table 6.4 contains the results obtained by performing the same experiments, as applied to the other S.O.R. methods, using the two Overlapping Block S.O.R. methods.

Method	Mesh size	n_A	n_E	ω_e	ω_b
(2×1) Overlapping Block SOR	10	23	11	1.358	1.397
	20	62	24	1.204	1.653
	40	137	47	1.147	1.808
	60	200	67	1.159	1.858
(2×2) Overlapping Block SOR	10	18	9	1.321	1.309
	20	45	19	1.298	1.583
	40	99	38	1.239	1.767
	60	149	55	1.255	1.833

TABLE 6.4

These two overlapping block methods lack consistent ordering, and the effect that this has, can plainly be seen in Table 6.4. Apart from the number of iterations being excessively large, the value of ω_b obtained from (6.4.9) is grossly inaccurate. Although subsequent attempts to overlap blocks were more successful, the strategy was abandoned because of the inability to estimate ω satisfactorily.

The final schemes that we shall investigate are Line S.O.R. Methods which involve the solution of one or more complete lines of mesh points by a direct method.

Consider the j^{th} column of mesh points (j=1,2,...,m). If equation (6.2.6) is applied to each mesh point in the column, then the following tridiagonal system of equations is created:-

$$\begin{bmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & -1 & 4 & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 4 \end{bmatrix} \begin{bmatrix} \phi_{1,j} \\ \phi_{2,j} \\ \vdots \\ \phi_{m,j} \end{bmatrix}^{(n+1)} = \begin{bmatrix} d_{1,j} \\ d_{2,j} \\ \vdots \\ d_{m,j} \end{bmatrix}^{(n)} \qquad (6.6.16)$$

where, the right hand side, $d_{i,j}$ (i=1,2,...,m), which consists of values at neighbouring mesh points not lying on the column, is defined as,

$$\left.\begin{aligned} d_{1,j}^{(n)} &= \phi_{1,j-1}^{(n+1)} + \phi_{0,j}^{(n+1)} + \phi_{1,j+1}^{(n)} \,, \\ d_{i,j}^{(n)} &= \phi_{i,j-1}^{(n+1)} + \phi_{i,j+1}^{(n)} \quad , \quad \text{for } i=2,3,\ldots,(m-1) \\ \text{and} \quad d_{m,j}^{(n)} &= \phi_{m,j-1}^{(n+1)} + \phi_{m+1,j}^{(n)} + \phi_{m,j+1}^{(n)} \,. \end{aligned}\right\} \qquad (6.6.17)$$

Clearly, these formulae represent the Gauss-Seidel Iteration scheme and one iteration involves the solution of m such systems of equations.

In order to perform this method on a parallel computer, there are two possible approaches. One approach involves the use of the Folding

Triangular Factorisation Method described in Chapter 3, i.e., each system of equations would be solved using this method thus permitting the use of two processors. Alternatively, Gaussian Elimination [Wilkinson, 1965] can be used to solve each system of equations, each processor of the parallel computer executing the algorithm on one or more of the m systems.

Since the properties of the Folding Triangular Factorisation Method are detailed in Chapter 3, we shall only consider the latter method. If the Gaussian Elimination Method is applied to the system of equations (6.6.16), the following set of equations are derived:

$$\left.\begin{aligned} & g_1 = \frac{1}{4} \qquad h_1 = d_{1,j}^{(n)} g_1 \\ & \left.\begin{aligned} g_i &= \frac{1}{(4-g_{i-1})} \\ h_i &= (d_{i,j}^{(n)} + h_{i-1}) g_i \end{aligned}\right\} \text{ for } i=2(1)m \ , \\ & \text{then, } \phi_{m,j}^{(n+1)} = h_m \\ & \text{and} \qquad \phi_{i,j}^{(n+1)} = \phi_{i,j}^{(n)} + g_i \phi_{i+1,j}^{(n+1)} \quad \text{for } i=m-1(-1)1 \end{aligned}\right\} \tag{6.6.18}$$

Now considering the S.O.R. method, the $(n+1)^{th}$ iterates of the j^{th} column are redefined by the equations,

$$\phi_{i,j}^{(n+1)} = \phi_{i,j}^{(n)} + \omega(\phi_{i,j}^{*} - \phi_{i,j}^{(n)}) \quad \text{for } i=1(1)m \ , \tag{6.6.19}$$

where $\phi_{i,j}^{*}$ $(i=1,2,\ldots,m)$ represents the Gauss-Seidel solution to the system of equations (6.6.16) which is defined in (6.6.18) as $\phi_{i,j}^{(n+1)}$. It is important to remember that the g_i $(i=1,2,\ldots,m)$ need only be calculated once since they remain constant for each system of equations and each iteration.

As with the point S.O.R. schemes and for the same reasons, it is necessary to consider the columns of mesh points in a red-black order. If the columns are numbered 1 to m from left to right, then each

iteration consists of two passes, the first of which includes columns 1,3,5...(m-1) and the second, columns 2,4,...m, assuming that m is even. Furthermore, the right hand side vector $\underline{d}^{(n)}$, is redefined for the first pass as,

$$\left.\begin{aligned} d^{(n)}_{1,j} &= \phi^{(n)}_{1,j-1} + \phi^{(n)}_{0,j} + \phi^{(n)}_{1,j+1} \\ d^{(n)}_{i,j} &= \phi^{(n)}_{i,j-1} + \phi^{(n)}_{i,j+1}\ , \quad \text{for } i=2,3,\ldots(m-1) \\ d^{(n)}_{m,j} &= \phi^{(n)}_{m,j-1} + \phi^{(n)}_{m+1,j} + \phi^{(n)}_{m,j+1} \end{aligned}\right\} \qquad (6.6.20)$$

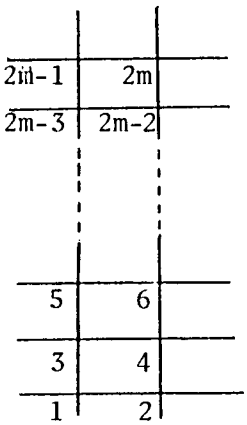
and for the second pass, the right hand side vector will be $\underline{d}^{(n+1)}$ as defined in (6.6.20) instead of $\underline{d}^{(n)}$ as defined in (6.6.17). Again, as with the point S.O.R. scheme, the first pass should be completed before the second is started but within each pass, the order in which the columns are considered or shared between the processors is not important. Of course, as with the point SOR schemes, it is also possible to apply the technique of folding to give a Folding Line S.O.R. method where, using two processors, one processor evaluates columns 1,3,5...(m-1) while the other evaluates columns m,m-2,...,2. Once more the right hand side is redefined as $\underline{d}^{(n)}$ before the processors cross and $\underline{d}^{(n+1)}$ after they have crossed where $\underline{d}^{(n)}$ is as in (6.6.20).

For both methods, assuming m is even and two processors are used, the number of parallel operations per iteration will be,

$$\frac{m}{2}(3m-1) \text{ multiplications} + \frac{5m^2}{2} \text{ additions}\ , \qquad (6.6.21)$$

with an additional m multiplications +(m-1) additions before the first iteration to evaluate the g_i (i=1,2,...,m).

Now consider two adjacent columns of mesh points which are numbered in the following way,



(6.6.22)

As with the single line method, equation (6.2.6) may be applied to each mesh point in the two columns to give the system of equations,

$$\begin{bmatrix} 4 & -1 & -1 & & & & & & & \\ -1 & 4 & 0 & -1 & & & & & & \\ -1 & 0 & 4 & -1 & -1 & & & & & \\ & -1 & -1 & 4 & 0 & -1 & & & & \\ & & -1 & 0 & & & \ddots & & & \\ & & & -1 & & & & -1 & & \\ & & & & \ddots & & & -1 & & \\ & & & & & -1 & & & -1 & \\ & & & & & & -1 & -1 & 4 \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{2m} \end{bmatrix}^{(n+1)} \quad \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{2m} \end{bmatrix}^{(n)} , \qquad (6.6.23)$$

where the solution vector $\underline{\phi}^{(n+1)}$ is defined as:

$$\underline{\phi}_k^{(n+1)} = \begin{cases} \phi_{(k+1)/2,j}^{(n+1)} , & \text{for } k=1,3,\ldots,(2m-1) \\ \phi_{k/2,j+1}^{(n+1)} , & \text{for } k=2,4,\ldots,2m \end{cases} \qquad (6.6.24)$$

and the right hand side vector, $\underline{d}^{(n)}$, is defined as,

$$\left.\begin{aligned} d_1^{(n)} &= \phi_{1,j-1}^{(n+1)} + \phi_{0,j}^{(n)} , \\ d_2^{(n)} &= \phi_{0,j+1}^{(n)} + \phi_{1,j+2}^{(n)} , \\ d_k^{(n)} &= \begin{cases} \phi_{(k+1)/2,j-1}^{(n+1)} , & \text{for } k=3,5,\ldots,(2m-3) \\ \phi_{k/2,j+2}^{(n)} , & \text{for } k=4,6,\ldots,(2m-2) \end{cases} \\ d_{2m-1}^{(n)} &= \phi_{m,j-1}^{(n+1)} + \phi_{m+1,j}^{(n)} \\ \text{and} \qquad d_{2m}^{(n)} &= \phi_{m+1,j+1}^{(n)} + \phi_{m,j+2}^{(n)} . \end{aligned}\right\} \qquad (6.6.25)$$

These formulae again represent the Gauss-Seidel Iteration scheme and one iteration involves the solution of m/2 such systems of equations.

The coefficient matrix is quindiagonal and so (6.6.23) may be solved using the Folding Triangular Factorisation Method or by Gauss Elimination. As with the single line S.O.R. method, the latter algorithm is considered which leads to the following formulae,

$$\left.\begin{aligned} & g_1 = 1/4, \quad a_1 = -g_1 \quad h_1 = d_1 g_1, \\ & g_2 = 1/(4+a_1), \quad a_2 = -g_1 g_2, \quad h_2 = (d_2+h_1)g_2, \end{aligned}\right.$$

and for k=3(1)2m,

$$\left.\begin{aligned} b_k &= \begin{cases} a_{k-2} & \text{, when k is odd} \\ (a_{k-2}-1) & \text{, when k is even} \end{cases} \\ g_k &= 1/(4-g_{k-2}-b_k a_{k-1}), \\ a_k &= \begin{cases} (b_k g_{k-1}-1)g_k & \text{, when k is odd} \\ b_k g_{k-1} g_k & \text{, when k is even} \end{cases} \\ \text{and} \quad h_k &= (d_k+h_{k-2}-b_k h_{k-1})g_k \end{aligned}\right\} \quad (6.6.26)$$

then

$$\phi_{2m}^{(n+1)} = h_{2m},$$

$$\phi_{2m-1}^{(n+1)} = h_{2m-1} - a_{2m-1}\phi_{2m}^{(n+1)}$$

and

$$\phi_k^{(n+1)} = h_k - a_k\phi_{k+1}^{(n+1)} + g_k\phi_{k+2}^{(n+1)}, \text{ for } k=2m-2,2m-3,\ldots 1.$$

As with the single line S.O.R. method, the application of the relaxation technique leads to the $(n+1)^{th}$ iterates of the j^{th} and $(j+1)^{th}$ columns being redefined to give the S.O.R. formulae as

$$\left.\begin{aligned} \phi_{i,j}^{(n+1)} &= \phi_{i,j}^{(n)} + \omega(\phi_{2i-1}^{*} - \phi_{i,j}^{(n)}) \\ \text{and} \quad \phi_{i,j+1}^{(n+1)} &= \phi_{i,j+1}^{(n)} + \omega(\phi_{2i}^{*} - \phi_{i,j+1}^{(n)}) \end{aligned}\right\} \text{for } i=1,2,\ldots m, \quad (6.6.27)$$

where $\underline{\phi}^*$ represents the Gauss-Seidel solution $\underline{\phi}^{(n+1)}$ defined in (6.6.26).

Again it is only necessary to calculate g_k, a_k and b_k (k=1,2,...2m) once. Obviously, with this 2 line S.O.R. method, m must be even and for

the same reasons as before, a red-black ordering will be used on the pairs of columns. This means that the right hand side vector $\underline{d}^{(n)}$ is now defined as

$$\left.\begin{aligned} d_1^{(n)} &= \phi_{1,j-1}^{(n)} + \phi_{0,j}^{(n)} , \\ d_2^{(n)} &= \phi_{0,j+1}^{(n)} + \phi_{1,j+2}^{(n)} , \\ d_k^{(n)} &= \begin{cases} \phi_{(k+1)/2,j-1}^{(n)} & , \text{ for } k=3,5,\ldots(2m-3) \\ \phi_{k/2,j+2} & , \text{ for } k=4,6,\ldots(2m-2) \end{cases} \\ d_{2m-1}^{(n)} &= \phi_{m,j-1}^{(n)} + \phi_{m+1,j}^{(n)} \\ \text{and} \quad d_{2m}^{(n)} &= \phi_{m+1,j+1}^{(n)} + \phi_{m,j+2}^{(n)} , \end{aligned}\right\} \qquad (6.6.28)$$

during the first pass and as $d^{(n+1)}$ during the second pass. Now, if m is divisible by 4, the number of parallel operations per iteration using 2 processors will be,

$$\frac{m}{4}(10m-5) \text{ multiplications} + \frac{m}{2}(6m-1) \text{ additions}, \qquad (6.6.29)$$

with an additional (8m-5) multiplications + (6m-3) additions before the first iteration to evaluate g_k, a_k and b_k (k=1,2,...2m).

Thus, we have defined 3 line SOR methods which we shall call the Line SOR, Folding Line SOR, and Two Line SOR Methods respectively. In order to compare the rates of convergence of these methods, the previously described experiments were applied to the methods the results of which are contained in Table 6.5.

It is clear from this table that there is a discrepancy between the estimated and actual values of n and ω for the Folding Line SOR Method. This is because, as with the Folding Point SOR Method, the sequential consistent ordering is not preserved. In this case, although not disastrous, the effect is noticeable. Not only is the rate of convergence less but the estimation of ω is inaccurate, and so the method will no

longer be considered but it serves to demonstrate the effect that the absence of consistent ordering can have.

Method	Mesh size	n_A	n_E	ω_e	ω_b
Line SOR	10	12	10	1.381	1.367
	20	22	22	1.630	1.625
	40	42	43	1.792	1.793
	60	61	61	1.855	1.849
Folding Line SOR	10	15	12	1.559	1.408
	20	30	23	1.731	1.640
	40	57	44	1.843	1.797
	60	81	62	1.888	1.850
Two Line SOR	10	9	8	1.273	1.244
	20	17	15	1.521	1.516
	40	31	31	1.729	1.724
	60	45	46	1.804	1.804

TABLE 6.5

The results obtained using the other two methods are more impressive and to make a direct comparison between them, the results contained in Table 6.5 must be combined with the numbers of parallel operations per iteration found in equations (6.6.21) and (6.6.29). This will give the total number of parallel operations required by each of the methods and these appear in Table 6.6.

Clearly, from Table 6.6, it can be seen that the single line SOR method, although requiring more additions, requires fewer multiplications than the two line SOR method. Since a multiplication operation takes longer than an addition operation, the line SOR method is faster operationally than the 2 line SOR method.

This line SOR technique can be extended to include more than two lines of mesh points. However, the resulting systems of equations will have a wider bandwidth and will be more sparse. Thus, the object of

Mesh size	10		20		40		60	
Method	M	A	M	A	M	A	M	A
Line SOR	$18m^2-5m$	$30m^2+m-1$	$33m^2-10m$	$55m^2+m-1$	$63m^2-20m$	$105m^2+m-1$	$\frac{183}{2}m^2-\frac{59}{2}m$	$\frac{305}{2}m^2+m-1$
Two Line SOR	$\frac{45}{2}m^2-\frac{13}{4}m-5$	$27m^2+\frac{3}{2}m-3$	$\frac{85}{2}m^2-\frac{53}{4}m-5$	$51m^2-\frac{5}{2}m-3$	$\frac{155}{2}m^2-\frac{123}{4}m-5$	$93m^2-\frac{19}{2}m-3$	$\frac{225}{2}m^2-\frac{193}{4}m-5$	$135m^2-\frac{33}{2}m-3$

where M=multiplications and A=additions

TABLE 6.6

using an iterative method to solve the original system of difference equations will have been defeated in that, once again, the method of solution will not take advantage of the large number of zeros that occur in the systems of equations.

In this section point, block and line SOR methods have been considered with some interesting results and so, to complete the analysis, a comparison will be made between these different types of method in the final section.

6.7 CONCLUSIONS

In this chapter, a variety of SOR methods have been investigated with some impressive results. It has become clear, however, that the way in which the system of mesh points is partitioned is not as important as the order in which values at the mesh points or blocks of mesh points are evaluated. Furthermore, the best ordering is the red-black ordering, since it produces the conditions necessary for a parallel algorithm.

The final choice of which method to use will depend on the characteristics of the parallel computer on which it is to be implemented such as the number of processors that are available. It is still useful to make some sort of comparison between the methods included here.

Let us consider the Red-Black Point, (2×1) Block, (2×2) Block, Line and Two Line SOR methods. Although the (2×2) Block Method requires 4 processors it is not difficult to implement it on a computer with 2 processors and so a comparison shall be made between the performances of the methods on a parallel computer with two processors. The most meaningful comparison that can be made concerns the total number of arithmetic operations required by each of the methods which is the product of the number of iterations and the number of operations per iteration. This technique has already been applied in the previous

section to some of the methods and, by applying it to the other methods, Table 6.7 can be produced.

Mesh size	10		20		40		60	
Method	M	A	M	A	M	A	M	A
Red-Black Point SOR	$22\frac{1}{2}m^2$	$37\frac{1}{2}m^2$	$43\frac{1}{2}m^2$	$72\frac{1}{2}m^2$	$85\frac{1}{2}m^2$	$142\frac{1}{2}m^2$	$123m^2$	$205m^2$
(2×1) Block SOR	$26m^2$	$45\frac{1}{2}m^2$	$52m^2$	$91m^2$	$100m^2$	$175m^2$	$146m^2$	$255\frac{1}{2}m^2$
(2×2) Block SOR	$27\frac{1}{2}m^2$	$49\frac{1}{2}m^2$	$55m^2$	$99m^2$	$105m^2$	$189m^2$	$152\frac{1}{2}m^2$	$274\frac{1}{2}m^2$
Line SOR*	$18m^2$	$30m^2$	$33m^2$	$55m^2$	$63m^2$	$105m^2$	$91\frac{1}{2}m^2$	$152\frac{1}{2}m^2$
Two Line* SOR	$22\frac{1}{2}m^2$	$27m^2$	$42\frac{1}{2}m^2$	$51m^2$	$77\frac{1}{2}m^2$	$93\ m^2$	$112\frac{1}{2}m^2$	$67\frac{1}{2}m^2$

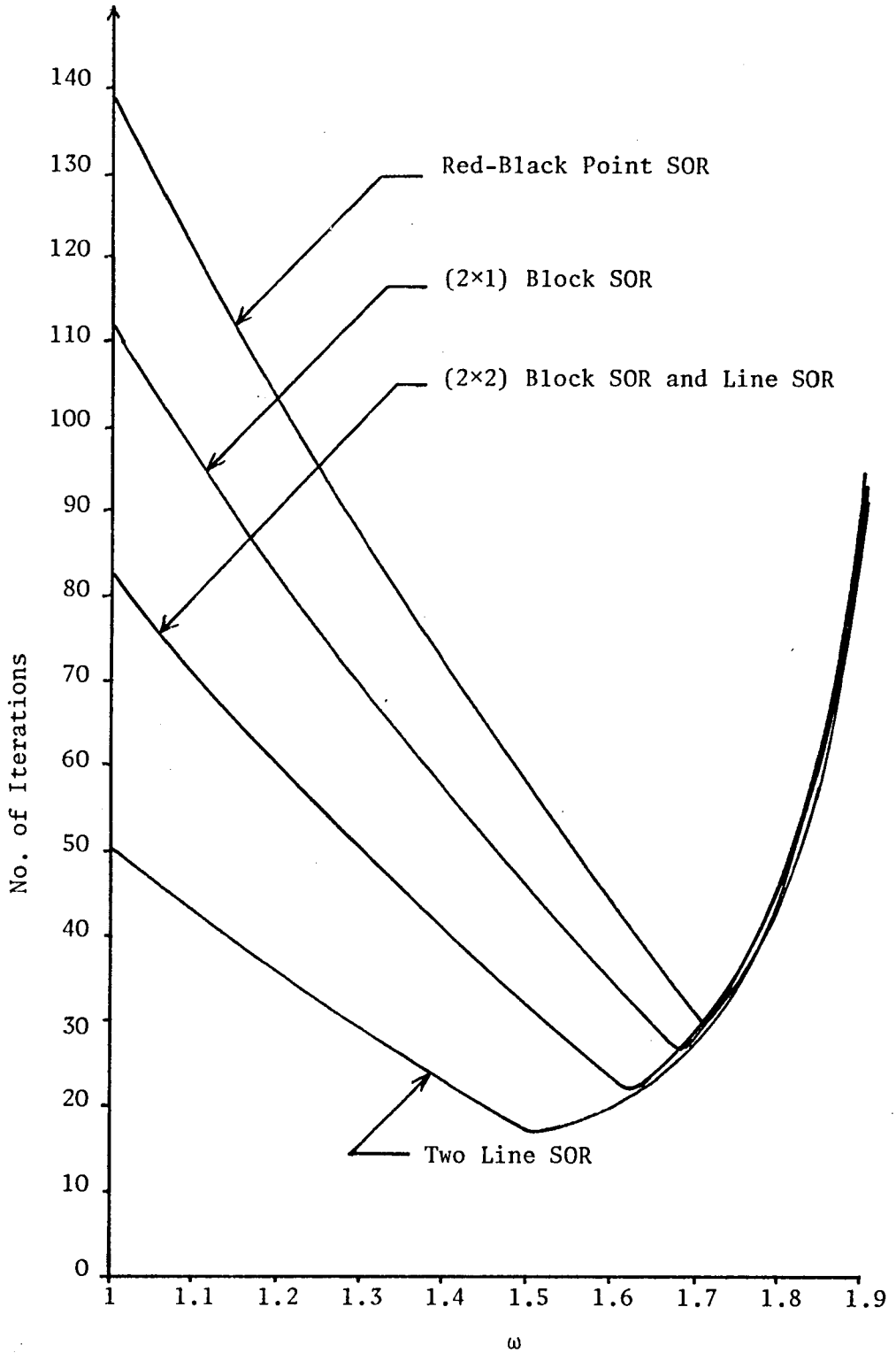
where M=multiplications and A=additions

**results on these lines include terms in m^2 only.*

TABLE 6.7

The results contained in Table 6.7 reveal that the Block SOR methods are not as good as the other methods. The best method is the Line SOR Method with the Two Line and Red-Black Point SOR Methods also achieving very good results. These results are not conclusive however since no allowance has been made for such factors as the unsolved problem of memory contention (see Chapter 2).

A final comment on the importance of consistent ordering arises from the effect that the choice of ω has on the number of iterations required for convergence of the method. The number of iterations against ω for a 20×20 system of meshes has been plotted in Graph 6.1 for the 5 methods considered in this section. Clearly, a bad choice of ω can greatly increase the number of iterations and so it is important to be able to estimate ω_b reasonably accurately.


140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
No. of Iterations
Red-Black Point SOR
(2×1) Block SOR
(2×2) Block SOR and Line SOR
Two Line SOR
1
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
ω

GRAPH 6.1

To conclude this chapter, let us reiterate the main points in developing a parallel SOR algorithm. Firstly, it is desirable but not vital to preserve the properties of the sequential ordering, i.e., property A and consistent ordering, and secondly, the ordering of the mesh points or blocks of mesh points is more important than the way in which the system of mesh points is partitioned. Finally the most useful ordering, although not the only one suitable, is red-black ordering.

N.B. The results included in this chapter, in particular in tables 6.1, 6.2, 6.4 and 6.5, were evaluated on the ICL 1904S computer at Loughborough University and rounded to three decimal figures.

CHAPTER 7

THE CORRECTION OF THE ELEMENTS OF THE INVERSE MATRIX BY IMPLICIT ITERATIVE PROCESSES

7.1 INTRODUCTION

The inversion of a matrix involves the solution of the matrix equation,

$$A.Y = I \quad , \tag{7.1.1}$$

for Y, the inverse of the $(m \times m)$ matrix A.

The computed solution Y of (7.1.1) may be improved or corrected by the application of the iterative procedure defined by the equations,

$$\left.\begin{aligned} R^{(n)} &= I - Y^{(n)}A \quad , \\ D^{(n)} &= R^{(n)}Y^{(n)} \quad , \\ \text{and} \qquad Y^{(n+1)} &= Y^{(n)} + D^{(n)} \quad , \end{aligned}\right\} \tag{7.1.2}$$

which can be reduced to the familiar explicit iterative process,

$$Y^{(n+1)} = Y^{(n)} + [I - Y^{(n)}.A]Y^{(n)} \quad , \tag{7.1.3}$$

where the initial approximation $Y^{(1)}$ is the computed solution Y of equation (7.1.1). Hotelling [1943] showed that this process has quadratic convergence.

In this chapter, two implicit methods are derived and are shown to have faster rates of convergence than (7.1.3). The first involves a similar amount of work and has the same rate of convergence as (7.1.3), but, by applying it in a different manner to that of (7.1.3), A^{-1} can be evaluated to the same degree of accuracy in a shorter time.

The second method is shown to have quartic convergence but despite the impressive results that it achieves, the excessive work that is involved per iteration makes it uncompetitive with other methods.

7.2 HOTELLING'S METHOD

The iterative process attributed to Hotelling is given in (7.1.3). It is essential that the right hand side of this equation is not expressed in the form $2Y^{(n)}-Y^{(n)}AY^{(n)}$ so as to avoid the cancellation of errors and thus make the method ineffective. It is also important that the residual or correction matrix,

$$\tilde{C}_n = (I - Y^{(n)}A) \quad , \tag{7.2.1}$$

is computed using the accumulation of inner products with each component of $\tilde{C}_n$ being rounded once on completion.

The iterative process is usually terminated when the following condition is satisfied

$$\text{max.abs.}[(\tilde{C}_n.Y^{(n)})_{i,j}]<2.\text{eps.}(\text{max.abs.}[Y^{(n)})_{i,j}]) \quad , \tag{7.2.2}$$

where $\text{max.abs.}[(Z)_{i,j}]$ is the greatest absolute value of the elements of matrix Z, and eps. is the smallest number for which 1+eps>1 on the given computer.

If we now consider the correction matrix $(I-Y^{(2)}A)$, then using (7.1.3) we have:

$$\begin{aligned}(I-Y^{(2)}A) &= I - (Y^{(1)} + (I-Y^{(1)}A)Y^{(1)})A \\ &= I - 2Y^{(1)}A + (Y^{(1)}A)^2 \\ &= (I-Y^{(1)}A)^2 = \tilde{C}_1^2 \quad .\end{aligned} \tag{7.2.3}$$

Hence we can say that, if all the roots of $\tilde{C}_1$ are less than unity in modulus, then the method converges quadratically.

7.3 THE DERIVATION OF IMPLICIT MATRIX PROCESSES

It is possible to derive implicit iterative methods by proceeding in the following manner. To avoid confusion, let $X^{(1)}$ be an approximation to the inverse of A. Then we can write,

$$X^{(1)}A = -L_1 + D_1 - U_1 \quad , \tag{7.3.1}$$

where L_1 and U_1 are strictly lower and upper triangular matrices respectively whose non-zero elements are small and D_1 is a diagonal matrix whose non-zero elements are usually close to unity.

If we pre-multiply (7.3.1) by D_1^{-1} we have,

$$D_1^{-1}X^{(1)}A = -D_1^{-1}L_1 + I - D_1^{-1}U_1 \quad , \tag{7.3.2}$$

and letting,

$$\left.\begin{aligned} \tilde{L}_1 &= D_1^{-1}L_1 \\ \text{and} \qquad \tilde{U}_1 &= D_1^{-1}U_1 \quad , \end{aligned}\right\} \tag{7.3.3}$$

we may rewrite (7.3.2) as,

$$D_1^{-1}X^{(1)}A = -\tilde{L}_1 + I - \tilde{U}_1 \quad , \tag{7.3.4}$$

which in turn may be rewritten in the alternative form,

$$\begin{aligned} D_1^{-1}X^{(1)}A &= (I-\tilde{L}_1)(I-\tilde{U}_1) - \tilde{L}_1\tilde{U}_1 \\ &= (I-F_1)G_1 \quad , \end{aligned} \tag{7.3.5}$$

where

$$\left.\begin{aligned} F_1 &= \tilde{L}_1\tilde{U}_1(I-\tilde{U}_1)^{-1}(I-\tilde{L}_1)^{-1} \\ \text{and} \qquad G_1 &= (I-\tilde{L}_1)(I-\tilde{U}_1) \quad . \end{aligned}\right\} \tag{7.3.6}$$

Equation (7.3.5) may be rearranged to have the form,

$$A^{-1} = G_1^{-1}(I-F_1)^{-1}D_1^{-1}X^{(1)} \quad , \tag{7.3.7}$$

which leads to the implicit matrix equation,

$$G_1X^{(2)} \approx (I-F_1)^{-1}D_1^{-1}X^{(1)} \quad , \tag{7.3.8}$$

where $X^{(2)}$ is a closer approximation to A^{-1} than $X^{(1)}$.

Now the elements of the matrix F_1 are certainly small from our initial assumptions and if all the roots of F_1 lie within the unit circle, we can expand $(I-F_1)^{-1}$ in the form of an infinite series in F_1. Thus

$$G_1X^{(2)} \approx (I+F_1+\ldots)D_1^{-1}X^{(1)} \quad . \tag{7.3.9}$$

If we now neglect all of the terms in F_1 we obtain the simple matrix equation,

$$G_1X^{(2)} \equiv (I-\tilde{L}_1)(I-\tilde{U}_1)X^{(2)} = D_1^{-1}X^{(1)} \quad . \tag{7.3.10}$$

Using (7.3.3) we may rewrite this as,

$$(I-D_1^{-1}L_1)(I-D_1^{-1}U_1)X^{(2)} = D_1^{-1}X^{(1)} \quad ,$$

which leads to,

$$(D_1-L_1)D_1^{-1}(D_1-U_1)X^{(2)} = X^{(1)} \quad . \tag{7.3.11}$$

Introducing the auxiliary matrix Y such that,

$$Y = D_1^{-1}(D_1-U_1)X^{(2)} \quad , \tag{7.3.12}$$

equation (7.3.11) may be expressed in the form,

$$\left.\begin{aligned} &(D_1-L_1)Y = X^{(1)} \\ \text{and} \quad &(D_1-U_1)X^{(2)} = D_1Y \end{aligned}\right\} \tag{7.3.13}$$

The matrices D_1,L_1 and U_1 may be easily obtained from (7.3.1) and thus so may (D_1-L_1) and (D_1-U_1). Hence, equations (7.3.13) may be solved by carrying out simple consecutive implicit forward and backward substitution processes acting on each column of $X^{(2)}$ using the corresponding column of $X^{(1)}$ as the right hand side vector.

Thus we have a first order implicit iterative method for improving the inverse of the matrix A.

A second order implicit process may be obtained by returning to equation (7.3.9) and this time retaining the initial term in the expansion of $(I-F_1)^{-1}$. Thus,

$$G_1X^{(2)} = (I+F_1)D_1^{-1}X^{(1)} \quad , \tag{7.3.14}$$

but from equation (7.3.5) we have,

$$F_1 = I-D_1^{-1}X^{(1)}AG_1^{-1} \quad ,$$

from which on substituting into (7.3.14) we obtain,

$$\begin{aligned} G_1X^{(2)} &= D_1^{-1}X^{(1)} + [I-D_1^{-1}X^{(1)}AG_1^{-1}]D_1^{-1}X^{(1)} \\ &= D_1^{-1}[X^{(1)}+[I-X^{(1)}AG_1^{-1}D_1^{-1}]X^{(1)}] \quad . \end{aligned} \tag{7.3.15}$$

Now from (7.3.6) we have,

$$\begin{aligned} G_1^{-1} &= [(I-\tilde{L}_1)(I-\tilde{U}_1)]^{-1} \\ &= (I-\tilde{U}_1)^{-1}(I-\tilde{L}_1)^{-1} \quad , \end{aligned}$$

and using (7.3.3),

$$G_1^{-1} = (D_1-U_1)^{-1}D_1(D_1-L_1)^{-1}D_1 \quad . \tag{7.3.16}$$

Substituting this result into (7.3.15) and rearranging we have

$$\begin{aligned}(D_1-L_1)D_1^{-1}(D_1-U_1)X^{(2)} &= X^{(1)} + [I-X^{(1)}A(D_1-U_1)^{-1}D_1(D_1-L_1)^{-1}]X^{(1)} \\ &= X^{(1)} + H_1X^{(1)} \quad ,\end{aligned} \tag{7.3.17}$$

where

$$H_1 = [I-X^{(1)}A(D_1-U_1)^{-1}D_1(D_1-L_1)^{-1}] \quad .$$

As with the first order process (7.3.11), the second order process may be expressed in the form

$$\left.\begin{aligned}&(D_1-L_1)Y = X^{(1)} + H_1X^{(1)} \\ \text{and}\quad &(D_1-U_1)X^{(2)} = D_1Y \quad ,\end{aligned}\right\} \tag{7.3.18}$$

where $Y = D_1^{-1}(D_1-U_1)X^{(2)}$,

which may be solved by carrying out consecutive forward and backward substitution processes acting on each column of $X^{(2)}$ and the corresponding column of $X^{(1)}$.

Again the matrices $D_1, L_1, U_1, (D_1-L_1)$ and (D_1-U_1) are easily obtainable and since the latter two are triangular in form, their inversion presents no special computational difficulties. Once the matrix product $X^{(1)}A$ has been evaluated, matrix H_1 is produced by first evaluating $(D_1-U_1)^{-1}$ and $D_1(D_1-L_1)^{-1}$, then their product $(D_1-U_1)^{-1}D_1(D_1-L_1)^{-1}$ and finally $[I-X^{(1)}A(D_1-U_1)^{-1}D_1(D_1-L_1)^{-1}]$. The greater potential of this second order process is seen in section 7.5.

7.4 CONVERGENCE PROPERTIES OF THE FIRST ORDER IMPLICIT PROCESS

Let us now consider the error of $X^{(1)}A$ which is quite simply the amount by which $X^{(1)}A$ differs from the true solution I. Thus we have,

$$I-X^{(1)}A = L_1 + U_1 + (I-D_1) = C_1 \quad . \tag{7.4.1}$$

When equation (7.3.10) is post-multiplied by A, we obtain the result,

$$G_1X^{(2)}A = D_1^{-1}X^{(1)}A$$

or $$X^{(2)}A = G_1^{-1}D_1^{-1}X^{(1)}A \quad . \tag{7.4.2}$$

Combining this result with equation (7.3.5), we obtain,

$$\begin{aligned} X^{(2)}A &= G_1^{-1}(I-F_1)G_1 \\ &= I-G_1^{-1}F_1G_1 \quad . \end{aligned} \tag{7.4.3}$$

Hence, the error of $X^{(2)}A$ is given by,

$$I - X^{(2)}A = G_1^{-1}F_1G_1 \quad , \tag{7.4.4}$$

where from equations (7.3.6) we have defined

$$F_1G_1 = \tilde{L}_1\tilde{U}_1$$

and $$G_1^{-1} = (I-\tilde{U}_1)^{-1}(I-\tilde{L}_1)^{-1} \quad .$$

Since the elements of $\tilde{L}_1$ and $\tilde{U}_1$ are small, and if the roots of L_1 and U_1 lie within the unit circle, then equation (7.4.4) may be expanded in the form,

$$\begin{aligned} I - X^{(2)}A &= (I-\tilde{U}_1)^{-1}(I-\tilde{L}_1)^{-1}\tilde{L}_1\tilde{U}_1 \\ &= (I+\tilde{U}_1+\ldots)(I+\tilde{L}_1+\ldots)\tilde{L}_1\tilde{U}_1 \\ &= \tilde{L}_1\tilde{U}_1 + (\tilde{L}_1+\tilde{U}_1)\tilde{L}_1\tilde{U}_1 + \tilde{U}_1\tilde{L}_1^2\tilde{U}_1+\ldots \qquad (7.4.5) \\ &= D_1^{-1}L_1D_1^{-1}U_1+ \ldots \quad . \qquad (7.4.6) \end{aligned}$$

At this stage it is necessary to discuss the order of magnitude of the elements of the matrices L_1,U_1 and D_1, which we would normally find in practice.

If, for example the first approximation to the inverse has been determined by a simple direct method, the elements of L_1,U_1 and $(I-D_1)$ will in general be of the same order of magnitude. Then, we may write,

$$L_1 = U_1 = D_1-I = O(\varepsilon) \quad . \tag{7.4.7}$$

Also, since matrix D_1 is a close approximation to I then its elements and likewise the elements of D_1^{-1} will have values close to unity and so may be replaced by I.

Hence, by applying norms to equation (7.4.6) to the first order of approximation we have,

$$N(I-X^{(2)}A) = N(L_1U_1) \quad , \tag{7.4.8}$$

where
$$N(C) = [\Sigma\Sigma C_{i,j}^2]^{\frac{1}{2}} \quad , \tag{7.4.9}$$

and is defined as the square root of the sum of the products of its elements by their complex conjugates.

Since, from (7.4.1) we have,

$$C_1^2 = (L_1+U_1+I-D_1)^2 \quad ,$$

then
$$N(C_1^2) > N(L_1U_1) \quad , \tag{7.4.10}$$

for omitted terms can only be positive or zero in the trivial case. Hence,from (7.2.3) we can say that,

$$N(I-X^{(2)}A) < N(I-Y^{(2)}A) \ . \tag{7.4.11}$$

Thus, the proposed implicit method has quadratic convergence and competes with Hotelling's method. However, it is obvious that the formula given by equations (7.3.13) converges faster through having smaller neglected terms. The amount of work in each iteration remains the same as that for the Hotelling's formula, i.e. $O(2m^3)$ multiplications per iteration where m is the order of the matrix.

We shall now derive an upper bound for the error in $X^{(i)}$ in terms of $N(X^{(1)})$ and $N(C_1)$, the norms of the matrices $X^{(1)}$ and C_1 respectively.

Let R_i denote the residual matrix. Then by definition we have,

$$R_i = I - X^{(i)}A \ , \text{ for } i=1,2,3,\ldots \ , \tag{7.4.12}$$

where
$$G_iX^{(i+1)} = D_iX^{(i)} \quad , \tag{7.4.13}$$

together with,

$$X^{(i)}A = -L_i+D_i-U_i \quad \text{and} \quad G_i = D_i^{-1}(D_i-L_i)D_i^{-1}(D_i-U_i) \ . \tag{7.4.14}$$

Now, by taking norms in equation (7.4.12), we have

$$N(R_1) = N(I-X^{(1)}A) = N(C_1)$$

and from equation (7.4.11),

$$N(R_2) = N(I-X^{(2)}A) < N(C_1^2) \ .$$

finally giving,

$$N(R_{i+1}) = N(I-X^{(i+1)}A) < N((C_1^2)^i) \quad . \tag{7.4.15}$$

Hence, it follows immediately that

$$N(A^{-1}-X^{(i+1)}) < N(C_1^{2^i}A^{-1}) \quad ,$$

which on substitution for A^{-1} as given by equation (7.4.1) yields the final required result

$$N(A^{-1}-X^{(i+1)}) < N(C_1^{2^i}(I-C_1)^{-1}X^{(1)})$$

$$< N(C_1^{2^i}(I+C_1+C_1^2+\ldots)X^{(1)}) \quad . \tag{7.4.16}$$

If $N(C_1) \leq k < 1$, the roots of C_1 are less than unity in absolute value and we obtain the result,

$$N(A^{-1}-X^{(i+1)}) < N(X^{(1)})k^{2^i}/(1-k) \tag{7.4.17}$$

$$\text{where } N(C_1) = k < 1 \quad .$$

This gives an upper bound for the difference between each element of $X^{(i+1)}$ and the corresponding element of A^{-1}.

A simpler limit can be derived when we use the relation

$$N(X^{(1)}) < mx \quad ,$$

where x is the greatest absolute value of any element of $X^{(1)}$ and we substitute this relation into equation (7.4.17).

7.5 CONVERGENCE PROPERTIES OF THE SECOND ORDER IMPLICIT PROCESS

We shall now apply a similar analysis to the second order implicit process as has been applied to the first order process in section (7.4). First of all we shall consider the error of $X^{(2)}A$ which, as before, is the amount by which $X^{(2)}A$ differs from I.

So, from equation (7.3.14) we have,

$$G_1X^{(2)} = (I+F_1)D_1^{-1}X^{(1)} \quad ,$$

which on post-multiplying by A and rearranging gives,

$$X^{(2)}A = G_1^{-1}(I+F_1)D_1^{-1}X^{(1)}A \quad . \tag{7.5.1}$$

Now, using the result of equation (7.3.5), we have

$$X^{(2)}A = G_1^{-1}(I-F_1^2)G_1$$
$$= I-G_1^{-1}F_1^2G_1 \quad . \tag{7.5.2}$$

Thus, the error in $X^{(2)}A$ is given by,

$$I - X^{(2)}A = G_1^{-1}F_1^2G_1 \quad . \tag{7.5.3}$$

Since, from the definitions of F_1 and G_1 in (7.3.6), we have,

$$F_1G_1 = \tilde{L}_1\tilde{U}_1$$

and

$$G_1^{-1}F_1 = (I-\tilde{U}_1)^{-1}(I-\tilde{L}_1)^{-1}\tilde{L}_1\tilde{U}_1(I-\tilde{U}_1)^{-1}(I-\tilde{L}_1)^{-1}$$
$$= (I+\tilde{U}_1+\ldots)(I+\tilde{L}_1+\ldots)\tilde{L}_1\tilde{U}_1(I+\tilde{U}_1+\ldots)(I+\tilde{L}_1+\ldots),$$

then,

$$I - X^{(2)}A = (\tilde{L}_1\tilde{U}_1)^2 + \text{higher powers of } \tilde{L}_1\tilde{U}_1$$
$$= (D_1^{-1}L_1D_1^{-1}U_1)^2 + \text{higher powers of } D_1^{-1}L_1D_1^{-1}U_1. \tag{7.5.4}$$

If we again examine the elements of the matrices L_1, U_1 and D_1, then (7.5.4) may be reduced to the form,

$$I - X^{(2)}A = (L_1U_1)^2 + \text{higher powers of } L_1U_1 \quad . \tag{7.5.5}$$

In the light of earlier observations concerning the error in the first order implicit process, we can write immediately that

$$N(I-X^{(2)}A) < N(C_1^4) \quad . \tag{7.5.6}$$

Thus the proposed method has quartic convergence and so represents an extremely powerful process. Unfortunately, offset against this advantage, we note that the computational requirements for the implicit formula is $O(\frac{16}{3}m^3)$ multiplications for each iteration.

As with the first order implicit method, we shall now derive an upper bound for the error in $X^{(i)}$ in terms of $N(X^{(1)})$ and $N(C_1)$.

Once again we have the residual matrix R_i defined in (7.4.12) as

$$R_i = I - X^{(i)}A \quad , \text{ for } i=1,2,3,\ldots \ ,$$

where now,

$$G_iX^{(i+1)} = D_iX^{(i)} + [I-D_i^{-1}X^{(i)}AG_i^{-1}]D_i^{-1}X^{(i)} \ ,$$

and taking norms we have,

$$N(R_1) = N(I-X^{(1)}A) = N(C_1) \ .$$

From equation (7.5.6) we now have

$$N(R_2) = N(I-X^{(2)}A) < N(C_1^4) \quad , \tag{7.5.7}$$

and hence,

$$N(R_{i+1}) = N(I-X^{(i+1)}A) < N((C_1^4)^i) \ . \tag{7.5.8}$$

Again it follows that

$$\begin{aligned} N(A^{-1}-X^{(i+1)}) &< N(C_1^{4^i}A^{-1}) \\ &= N(C_1^{4^i}(I-C_1)X^{(1)}) \\ &< N(C_1^{4^i}(I+C_1+C_1^2+\ldots)X^{(1)}) \ , \end{aligned} \tag{7.5.9}$$

and $N(C_1) \leq k<1$, the roots of C_1 are less than unity in absolute value and we obtain the result,

$$N(A^{-1}-X^{(i+1)}) < N(X^{(1)})k^{4^i}/(1-k) \tag{7.5.10}$$

where $N(C_1)=k<1$.

Thus, as in (7.4.17), we have produced an upper bound for the error in $X^{(i+1)}$ and again this may be simplified by the use of the relation

$$N(X^{(1)}) < mx,$$

where x is the greatest absolute value of any element of $X^{(1)}$.

7.6 IMPLEMENTATION OF IMPLICIT ITERATIVE METHODS AND RESULTS

In the following experiments we shall consider the inversion of symmetric positive definite matrices. The normal implementation of Hotelling's method involves the calculation of an initial approximation to A^{-1} by a direct method, i.e., Choleski factorisation (Martin, Peters and Wilkinson [1965], [1966]).

Choleski factorisation may be defined as follows:

$$A = L.L^T \tag{7.6.1}$$

where L is a lower triangular matrix whose elements are determined by the following formulae,

for i=1(1)n

$$\left.\begin{aligned} \ell_{i,j} &= (a_{i,j} - \sum_{k=1}^{j-1} \ell_{i,k}\ell_{j,k})/\ell_{j,j} \\ &\qquad\qquad \text{for } j=1,\ldots,i-1. \\ \text{and} \qquad \ell_{i,i} &= (a_{i,i} - \sum_{k=1}^{i-1} \ell_{i,k}^2)^{\frac{1}{2}} \end{aligned}\right\} \tag{7.6.2}$$

Since triangular matrices are easily invertable, then we may take advantage of this fact by observing that,

$$A^{-1} = (L.L^T)^{-1} = (L^T)^{-1}.L^{-1} = (L^{-1})^T L^{-1} \ . \tag{7.6.3}$$

If we denote L^{-1} (also a lower triangular matrix) by P, then the elements of P are defined thus,

for i=1(1)n

$$\left.\begin{aligned} p_{i,i} &= 1/\ell_{i,i} \\ p_{j,i} &= -(\sum_{k=i}^{j-1} \ell_{j,k}p_{k,i})/\ell_{j,j} \\ &\qquad\qquad \text{for } j=i+1,\ldots,n. \end{aligned}\right\} \tag{7.6.4}$$

Finally, we have

$$A^{-1} = P^T.P \ ,$$

and since A^{-1} is symmetric it is only necessary to calculate the lower triangle of elements defined as follows:

$$(A^{-1})_{j,i} = (P^T.P)_{j,i} = \sum_{k=j}^{n} p_{k,j}p_{k,i} \tag{7.6.5}$$

for j=i,...,n
and i=1,...,n.

Now using this method to provide an initial approximation to A^{-1}, the three methods i.e., Hotelling's method and the first and second order implicit iterative methods, were used to find the inverses of the following examples, taken from Gregory and Karney [1969]

Example 1

$$A = \begin{bmatrix} 1.0 & -0.02 & -0.12 & -0.14 \\ -0.02 & 1.0 & -0.04 & -0.06 \\ -0.12 & -0.04 & 1.0 & -0.08 \\ -0.14 & -0.06 & -0.08 & 1.0 \end{bmatrix}$$

Example 2 The quindiagonal matrix,

$$A = \begin{bmatrix} 5 & -4 & 1 & 0 & 0 \\ -4 & 6 & -4 & 1 & 0 \\ 1 & -4 & 6 & -4 & 1 \\ 0 & 1 & -4 & 6 & -4 \\ 0 & 0 & 1 & -4 & 5 \end{bmatrix}$$

Example 3 The Wilson matrix,

$$A = \begin{bmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{bmatrix}$$

Example 4

$$A = \begin{bmatrix} 10 & 1 & 4 & 0 \\ 1 & 10 & 5 & -1 \\ 4 & 5 & 10 & 7 \\ 0 & -1 & 7 & 9 \end{bmatrix}$$

Example 5

$$A = \begin{bmatrix} 420 & 210 & 140 & 105 \\ 210 & 140 & 105 & 84 \\ 140 & 105 & 84 & 70 \\ 105 & 84 & 70 & 60 \end{bmatrix}$$

The performance of each method is recorded in Table 7.1 where for each example we give the number of iterations required for convergence of the formula and the run time. The condition for convergence is given in equation (7.2.2).

In the first example, each method converges in one iteration. Clearly the initial approximation to A^{-1} is correct to within the accuracy of the computer and the first application of each iterative formula merely confirms this.

In examples 2,3,4 and 5, each method converges in two iterations. In these cases, the initial approximation to A^{-1} is very good and the first iteration produces the correct result to within machine accuracy which the second iteration confirms.

Obviously, the initial approximation to A^{-1} provided by the Choleski inversion method is too accurate to demonstrate the improved convergence rates of the implicit iterative methods, so the examples

were repeated using the identity matrix as the initial approximation to A^{-1}. The results from the second set of experiments are contained in Table 7.2.

The results for example 1 clearly demonstrate the different convergence rates. In complete agreement with sections 7.4 and 7.5, the first order implicit method has a slightly better convergence rate than Hotelling's method while the second order implicit method requires only half the number of iterations that Hotelling's method requires.

Hotelling's method does not converge for examples 2,3,4 and 5, but the implicit methods do. The second order method requires approximately the same number of iterations as the first order method. This is due to the effect of rounding errors in the second order method, which is as might be expected because of the extra arithmetic involved.

It is clear, from Tables 7.1 and 7.2, that Hotelling's method does not always converge when using I as an initial approximation to A^{-1} and that the more accurate approximation produced by the Choleski Inversion Method must be used. Using this approximation with the first order implicit method, it can be seen that it is competitive with Hotelling's method both in terms of the number of iterations and the run time, being at least as fast as Hotelling's method. However, since the first order method is reliable when using I as an initial approximation to A^{-1}, then unlike Hotelling's method it is not necessary to use the more accurate approximation. So, comparing the run times for the First Order Method using I with those of Hotelling's Method using Choleski Inversion, we see that the former method is significantly faster.

The second order implicit method, although achieving some impressive results in terms of the number of iterations, is relatively slow because of the excessive work involved per iteration.

EXAMPLE		HOTELLING'S METHOD	1ST ORDER IMPLICIT METHOD	2ND ORDER IMPLICIT METHOD
1	No. of iterations	1	1	1
	Run time	24	23	27
2	No. of iterations	2	2	2
	Run time	25	24	29
3	No. of iterations	2	2	2
	Run time	24	24	27
4	No. of iterations	2	2	2
	Run time	24	23	28
5	No. of iterations	2	2	2
	Run time	25	23	27

N.B. The run time is given in mill units

TABLE 7.1

EXAMPLE		HOTELLING'S METHOD	1ST ORDER IMPLICIT METHOD	2ND ORDER IMPLICIT METHOD
1	No. of iterations	6	5	3
	Run time	21	22	26
2	No. of iterations	Does not converge	6	6
	Run time		24	32
3	No. of iterations	Does not converge	6	5
	Run time		22	28
4	No. of iterations	Does not converge	6	6
	Run time		23	29
5	No. of iterations	Does not converge	6	5
	Run time		23	29

N.B. The run time is given in mill units

TABLE 7.2

Finally, the program for the first order implicit method calculates the full inverse A while the program for Hotelling's only produces the lower triangle of A^{-1}. Hence the first order method's program can be used to evaluate the inverse of an unsymmetric matrix while the program for Hotelling's method would have to be adapted to produce the full inverse of the matrix.

CHAPTER 8

CONCLUSIONS

In the opening chapters of this thesis, various types of parallel computer were discussed. It was stated that since the problems associated with SIMD computers were less formidable than those associated with MIMD computers, the present state of the development of SIMD computers is more advanced. However, it is also obvious that it is important not to neglect the development of MIMD computers since there is only a relatively small class of problems for which the solution on an SIMD computer is eminently suitable. The point is that it is important not to develop one type of computer while neglecting others, particularly if, for that type of computer, it is difficult or impossible to design good algorithms.

In the same way it is important not to develop one type of algorithm, while neglecting others, if there is not a suitable computer on which to use that type of algorithm. At the same time a type of algorithm should not be dismissed completely if it does not perform as well as another type of algorithm on existing computers.

Clearly then, it is important that the development of computers and algorithms goes hand in hand, i.e., the computer designer must be aware of the types of algorithms available while the algorithm designer must be aware of the capabilities and performance of the proposed computers.

The algorithms presented in this thesis outline a variety of different strategies for developing parallel algorithms. A general classification of the different types of parallel algorithm has been made by Kung [1976] and are as follows:

(a) synchronized parallel algorithms,

(b) asynchronous parallel algorithms,

(c) synchronized iterative algorithms,

(d) asynchronous iterative algorithms,

(e) semi-synchronized iterative algorithms,

and (f) adaptive asynchronous algorithms.

To define these classes of algorithms, it is assumed that an algorithm consists of segments some or all of which can be executed in parallel.

A parallel algorithm is said to be synchronised if one of the segments of the algorithm cannot be executed until one or more of the other segments have been completed. As an example, consider the evaluation of the simple expression,

$$T = A \times B + C \times D \times E \quad , \tag{8.1}$$

which has three segments, s_1, s_2 and s_3, defined as

$$\left.\begin{aligned} & s_1 \text{ is } X = A \times B \\ & s_2 \text{ is } Y = C \times D \times E \\ \text{and } & s_3 \text{ is } T = X + Y \end{aligned}\right\} \tag{8.2}$$

Clearly s_1 and s_2 can be executed concurrently, but s_3 cannot be started until s_1 and s_2 have been completed and so this is a simple synchronized parallel algorithm.

When there is no such dependency between segments, the parallel algorithm is said to be asynchronous. In general, with an asynchronous algorithm there will be global variables, accessible to all processors, which control which segment is executed next by a processor. The manipulation of global variables would be programmed as a critical section to protect the variables from being operated on by more than one processor simultaneously. A simple example of this type of algorithm is the addition of two vectors (2.1.1). If one segment is

$$\left.\begin{aligned} & c_i = a_i + b_i \quad , \quad \text{for } i=1,2,\ldots n/2 \ , \\ \text{and another is } \quad & c_j = a_j + b_j \quad , \quad \text{for } j=n/2,n/2+1,\ldots n, \end{aligned}\right\} \tag{8.3}$$

then obviously there is no dependency between the two segments.

In a synchronized iterative algorithm, each iteration has more than one segment and synchronization occurs at the end of each

iteration, i.e., the $(n+1)^{th}$ iteration cannot commence until the n^{th} iteration is completed. As an example, consider the Newton iteration formula

$$x_{i+1} = x_i - f'(x_i)^{-1} f(x_i) \quad , \tag{8.4}$$

which evaluates the zeros of function f. During each iteration f(x) and f'(x) can be evaluated followed by x_{i+1} which is where the synchronization is needed.

An asynchronous iterative algorithm does not require synchronization at all. This may be illustrated by using the same example as used for the synchronized iterative algorithm but with one processor evaluating f(x) and x, and another evaluating f'(x). If each processor uses the latest values of x, f(x) and f'(x), then they can run independently. This is then an asynchronous iterative algorithm.

Semi-synchronized iterative algorithms are a combination of synchronized and asynchronous iterative algorithms. If one processor is currently on its i^{th} iteration and another on its j^{th} iteration then, assuming that i>j, a restriction is imposed such that i-j<b, where b is a positive integer. This implies that the first processor does not get more than b iterations ahead of the second. This technique can easily be applied to the asynchronous Newton iteration algorithm.

Finally, there are adaptive asynchronous algorithms in which the number of segments performed by each processor are not pre-determined but depend on the relative speeds of the processors. Consider the example of vector addition (8.3) used to illustrate the asynchronous parallel algorithm. Although the work has been shared equally between the processors, if the processors do not work at the same speed, the difference in the time that each processor requires can be considerable. Let the two segments be redefined as

$$\left.\begin{aligned} c_i &= a_i + b_i \quad i=1,2,\ldots,m \\ \text{and} \quad c_j &= a_j + b_j \quad j=n,n-1,\ldots,m+1 \end{aligned}\right\} \tag{8.5}$$

where m is determined at run time i.e., each processor continues operating until the indices i and j are such that i=j =m. This is an adaptive asynchronous algorithm. Obviously, the time between each processor finishing can be at most equal to the time required to evaluate one element of $\underline{c}$ and so the algorithm is expected to be relatively efficient.

If we classify the main algorithms presented in this thesis using these definitions, the algorithms presented in Chapters 3 and 4 for the solution of banded and triangular systems of equations are synchronised parallel algorithms. The Parallel Quicksort Algorithm (Chapter 5) is an adaptive asynchronous algorithm and the S.O.R. strategies presented in Chapter 6 are synchronized iterative algorithms.

The implementation of the algorithms, contained in this thesis, on a parallel computer has been limited by both time and opportunity. The only available working computer is the Loughborough University Department of Computer Studies Interdata Dual Processor which has restricted experiments to the use of two processors only. However, some of the algorithms have been implemented successfully [Barlow, 1977(a) and (b), and Barlow and Evans,1977] and the Speed Ups attained by them are presented in Table 8.1.

Algorithm	Order of Problem	Speed Up	Efficiency
Parallel Triangular Factorisation Method (Chapter 3)	m=2 n=64	1.6	0.8
Parallel Quicksort Method (Chapter 5)	n=1024	1.6	0.8
Parallel Line S.O.R. Method (Chapter 6)	m=18 m=30	1.81 1.87	0.905 0.935

TABLE 8.1

These results are clearly very encouraging at this stage in the development of both parallel algorithms and the Interdata Dual Processor. The efficiency of the Parallel Triangular Factorisation and Parallel Quicksort Methods are not quite as good as the Parallel S.O.R. Method. This is due to synchronization in the former case, and in the latter case to the initial partitioning step being sequential. The results also give some idea of the parallel overheads which, although not large, are significant.

Conclusive evidence is not presented here of any type of algorithm being significantly better than others. Any such evidence would only be conclusive of course for the Interdata Dual Processor. However the results do reveal that for MIMD computers, it is best to keep sequential segments of a parallel algorithm to a minimum and avoid excessive synchronization. Thus, these two points coupled with the techniques applied in this thesis should enable the development of good parallel algorithms for the solution of many problems.

REFERENCES

AHO A.V., HOPCROFT J.E. and ULLMAN J.D. [1974], "The Design and Analysis of Computer Algorithms", Addison-Wesley.

ANDERSON J.P., [1965], *"Program Structures for Parallel Processing"*, Comm. ACM, Volume 8, pp.786-788.

BARLOW R.H., [1977a], *"Performance of a Dual-Minicomputer Parallel Computer System"*, Internal Report No. 43, Dept. of Computer Studies, Loughborough University.

BARLOW R.H., [1977b], *"Parallel Algorithms for Sorting, Quadrature and Eigenvalue Determination"*, Internal Report No. 44, Dept. of Computer Studies, Loughborough University.

BARLOW R.H. and EVANS D.J., [1977], *"An Analysis of the Performance of a Dual-Minicomputer Parallel Computer System"*, Internal Report No. 59, Dept. of Computer Studies, Loughborough University.

BARLOW et al [1977], Barlow R.H., Evans D.J., Newman I.A., Slade A.J. and Woodward M.C., *"Historical Survey of the Implementation of Parallel Programming on the Interdata Dual Processor Computer"*, Internal Report No. 40, Dept. of Computer Studies, Loughborough University.

BARNES et al [1968], Barnes G.H., Brown R.M., Kato M., Kuck D.J., Slotnick D.L. and Stoker R.A., *"The Illiac IV Computer"*, I.E.E.E. Trans. on Comp., Volume C-17, pp.746-757.

BATCHER K.E., [1968], *"Sorting Networks and their Applications"*,
Proc. AFIPS Spring Joint Comp. Conf., Volume 32, pp.307-314.

BORODIN A., [1971], *"Horner's Rule is Uniquely Optimal"*,
Theory of Machines and Computations, Kohavi Z. and Paz A. eds.,
Academic Press, N.Y., pp.45-58.

BORODIN A. and MUNRO I., [1975], *"The Computational Complexity of Algebraic and Numeric Problems"*,
American Elsevier, N.Y.

BOUKNIGHT et al, [1972], Bouknight W.J., Denenberg S.A., McIntyre J.M., Randall J.M., Sameh A.H. and Slotnick D.L., *"The Illiac IV System"*,
Proc. I.E.E.E., Volume 60, pp.369-388.

CARRÉ B.A., [1961], *"The Determination of the Optimum Acceleration Factor for Successive Over-relaxation"*,
Comp. J. 4, pp.73-78.

CHEN S.C. and KUCK D.J., [1975], *"Time and Parallel Bounds for Linear Recurrence Systems"*,
I.E.E.E. Trans. on Comp., Volume C-24, pp.701-717.

CSANKY L., [1975], *"Fast Parallel Matrix Inversion Algorithms"*,
Contributed paper, 16th Ann. Symp. on Foundations of Computer Science (SWAT), Berkeley.

DORN W.S., [1962], *"Generalizations of Horner's Rule for Polynomial Evaluation"*,
IBM J. of Res. and Devel., Volume 6, pp.239-245.

EVANS D.J. and Atkinson L.V., [1970], *"An Algorithm for the Solution of General Three Term Linear Systems"*,
Comp. J., Volume 13, pp.323-326.

EVANS D.J. and Hatzopoulos M., [1976], *"The Solution of Certain Banded Systems of Linear Equations using the Folding Algorithm"*,
Comp. J., Volume 19, pp.184-187.

FLYNN, M.J., [1966], *"Very High Speed Computing Systems"*,
Proc. of the I.E.E.E., Volume 54, pp.1901-1909.

GILMORE P.A., [1971], *"Parallel Relaxation"*,
Goodyear Aerospace Corp., Akron, Ohio.

GREGORY R.T. and KARNEY D.L. [1969]. *"A Collection of Matrices for Testing Computational Algorithms"*, Wiley Interscience.

HAYES L., [1974], *"Comparative Analysis of Iterative Techniques for solving Laplace's Equation on the unit square on a Parallel Processor"*,
M.Sc. Thesis, Dept. of Math., Univ. of Texas, Austin.

Heller D., [1974a], *"A Determinent Theorem with Applications to Parallel Algorithms"*,
SIAM J. Num. Anal., Volume 11, pp.559-568.

Heller D., [1974b], *"On the Efficient Computation of Recurrence Relations"*,
Report, I.C.A.S.E., NASA Langley Research Centre, Hampton, Virginia.

HELLER D., [1974c], *"Some Aspects of the Cyclic Reduction Algorithm for Block Tridiagonal Linear Systems"*,
I.C.A.S.E., Hampton Va.; Dept. of Comp. Sc., Carnegie-Mellon University.

HELLER D., [1976], *"A Survey of Parallel Algorithms in Numerical Linear Algebra"*,
Dept. of Comp. Sc., Carnegie-Mellon University.

HELLER D., STEVENSON D.K. and TRAUB J.F., [1974], *"Accelerated Iterative Methods for the Solution of Tridiagonal Linear Systems on Parallel Computers"*,
Dept. of Comp. Sc., Carnegie-Mellon University.

HINTZ R.G. and TATE D.P., [1972], *"Control Data STAR-100 Processor Design"*
COMPCON-72 Digest of Papers, I.E.E.E. Comp.Soc., pp.1-4.

HOARE C.A.R., [1962], *"Quicksort"*,
Comp. J., Volume 5, pp.10-15.

HOTELLING H., [1943], *"Some New Methods in Matrix Calculations"*,
Ann. Math. Stat., 14, pp.1-34.

HYAFIL L. and KUNG H.T., [1974], *"Parallel Algorithms for Solving Triangular Linear Systems with Small Parallelism"*,
Dept. of Comp. Sc., Carnegie-Mellon University.

HYAFIL L. and KUNG H.T., [1975], *"Bounds on the Speed-ups of Parallel Evaluation of Recurrences"*,
Second USA-Japan Comp. Conf. Proc. pp.178-182.

INTERDATA INC., [1971], Model 55: Dual Memory Bank Controller;
Information Specification, Interdata Inc., New Jersey 07757, U.S.A.

KNUTH D.E., [1973], *"Sorting and Searching"*,
The Art of Computer Programming, Volume 3, Addison-Wesley.

KOGGE P.M., [1972a], *"Parallel Algorithms for the Efficient Solution of Recurrence Problems"*,
Digital Systems Lab., Stanford University.

KOGGE P.M., [1972b], *"The Numerical Stability of Parallel Algorithms for Solving Recurrence Problems"*,
Digital Systems Lab., Stanford University.

KOGGE P.M., [1972c], *"Minimal Parallelism in the Solution of Recurrence Problems"*,
Digital Systems Lab., Stanford University.

KOGGE P.M., [1974], *"Parallel Solution of Recurrence Problems"*,
IBM J. of Res. and Devel., Volume 18, pp.138-148.

KOGGE P.M. and STONE H.S., [1973], *"A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations"*,
I.E.E.E. Trans. on Comp., Volume C-22, pp.786-793.

KUCK D.J., [1968], *"Illiac IV Software and Application Programming"*, I.E.E.E. Trans. on Comp., Volume C-17, pp.758-770.

KUCK D.J., [1973], *"Multi-operation Machine Computational Complexity"*, Complexity of Sequential and Parallel Numerical Algorithms, Traub J.F. ed., Academic Press, N.Y., pp.17-47.

KUCK D.J. and MARUYAMA K., [1975], *"Time Bounds on the Parallel Evaluation of Arithmetic Expressions"*, SIAM J. Comput., Volume 4, pp.147-162.

KUNG H.T., [1976], *"Synchronised and Asynchronous Parallel Algorithms for Multiprccessors"*, Dept. of Comp. Sc., Carnegie-Mellon University.

LAMBIOTTE J.J., [1975], *"The Solution of Linear Systems of Equations on a Vector Computer"*, Dissertation, University of Virginia.

LAMBIOTTE J.J. and VOIGT R.G., [1975], *"The Solution of Tridiagonal Linear Systems on the CDC STAR-100 Computer"*, A.C.M. Trans. on Math. Software, Volume 1, pp.308-329.

LAWRIE et al, [1975], Lawrie D.M., Layman T., Baer D. and Randal J.M., *"Glypnis, a Programming Language for Illiac IV"*, Comm. A.C.M., Volume 18, pp.157-164.

LIU, J.W.H., [1974], *"The Solution of Mesh Equations on a Parallel Computer"*, Dept. of Comp. Sc., University of Waterloo.

MARTIN R.S., PETERS G. and WILKINSON J.H., [1965], *"Symmetric Decomposition of a Positive Definite Matrix"*, Numer. Math. 7, pp.362-383, also Handbook for Automatic Computation, Volume 2, pp.9-30, Springer-Verlag (1971).

MARTIN R.S., PETERS G. and WILKINSON J.H., [1966], *"Iterative Refinement of the Solution of a Positive Definite System of Equations"*, Numer. Math. 8, pp.203-216, also Handbook for Automatic Computation, Volume 2, pp.31-44, Springer-Verlag (1971).

MARUYAMA K., [1973], *"The Parallel Evaluation of Matrix Expressions"*, IBM T.J. Watson Research Centre, Yorktown Heights, N.Y.

MIRANKER W.L., [1971], *"A Survey of Parallelism in Numerical Analysis"*, SIAM Review, Volume 13, pp.524-547.

MURAOKA Y. and KUCK D.J., [1973], *"On the Time Required for a Sequence of Matrix Products"*, Comm. ACM, Volume 16, pp.22-26.

NAG, [1976], NAG Library Manual, Mark 5, Volume 3 (Algol) NAG Limited, Oxford.

ORCUTT S.E., [1974], *"Parallel Solution Methods for Triangular Linear Systems of Equations"*,
Report 77, Digital Systems Lab., Stanford University.

OWENS J.L., [1973], *"The Influence of Machine Organisation on Algorithms"*,
Complexity of Sequential and Parallel Numerical Algorithms, Traub J.F. ed., Academic Press, N.Y., pp.111-130.

PARKINSON D., [1976], *"The ICL Distributed Array Processor - DAP"*,
Computational Methods in Classical and Quantum Physics, Hooper M.B. ed., Advance Publications Limited, pp.415-422.

PEASE M.C., [1967], *"Matrix Inversion using Parallel Processing"*,
J. ACM, Volume 14, pp.757-764.

PEASE M.C., [1968], *"An Adaption of the Fast Fourier Transform for Parallel Processing"*,
J. ACM, Volume 15, pp.252-264.

POOLE W.G. and VOIGT R.G., [1974], *"Numerical Algorithms for Parallel and Vector Computers: An annotated bibliography"*,
Computing Reviews, Volume 15, pp.379-388.

REDDAWAY S.F., [1973], *"DAP - A Distributed Array Processor"*,
1st Annual Symposium on Computer Architecture, Florida.

SAMEH A.H., [1971],*"On Jacobi and Jacobi-like Algorithms for Parallel Computers"*
Math. Comp. Volume 25, pp.579-590.

SAMEH A.H., CHEN S.C. and KUCK D.J., [1974], *"Parallel Direct Poisson and Biharmonic Solvers"*,
Dept. of Comp. Sc., University of Illinois, Urbana.

SAMEH A.H. and KUCK D.J., [1971], *"Parallel Computation of Eigenvalues of Real Matrices"*,
IFIP Congress 1971, North Holland, Amsterdam, Volume 2, pp.1266-1272.

SAMEH A.H. and KUCK D.J., [1975], *"Linear System Solvers for Parallel Computers"*,
Dept. of Comp. Sc., University of Illinois, Urbana.

SEDGEWICK R., [1975], *"Quicksort"*,
Ph.D. Thesis, Dept. of Comp. Sc., Stanford University.

SHELL D.L., [1959], *"A High-Speed Sorting Procedure"*,
Comm. of the ACM, Volume 2, pp.30-32.

SINGLETON R.C., [1969], *"An Efficient Algorithm for Sorting with Minimal Storage"*,
Comm. ACM, Volume 12, pp.185-187.

SMITH G.D., [1965], *"Numerical Solution of Partial Differential Equations"*,
Oxford University Press.

STEVENSON D.K., [1975], *"Programming the Illiac"*,
Dept. of Comp. Sc., Carnegie-Mellon University.

STONE H.S., [1971], *"Parallel Processing with the Perfect Shuffle"*, I.E.E.E. Trans. on Comp. Volume C-20, pp.153-161.

STONE H.S., [1973a], *"An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations"*, J. of the ACM, Volume 20, pp.27-38.

STONE H.S., [1973b], *"Problems of Parallel Computation"*, Complexity of Sequential and Parallel Numerical Algorithms, Traub J.F. ed., Academic Press, N.Y., pp. 1-16.

STONE H.S., [1975a], *"Parallel Tridiagonal Equation Solvers"*, ACM Trans. on Math. Software, Volume 1, pp.289-307.

STONE H.S., [1975b], *"Parallel Computers"*, Introduction to Computer Architecture, Stone H.S. ed., Science Research Associates, Palo Alto, California, pp.318-374.

VARGA R.S., [1962], *"Matrix Iterative Analysis"*, London Prentice-Hall International.

WATSON W.J., [1972], *"The T.I. ASC, a highly modular and flexible super-computer architecture"*, AFIPS Fall 1972, AFIPS Press, Montvale, N.J., Volume 41, Part 1, pp.221-229.

WICHMANN B.A., [1973], *"Algol 60 Compilation and Assessment"*, Academic Press, London and New York, pp.65-124.

WILKINSON J.H. [1963], *" Rounding Errors in Algebraic Processes"*, HMSO.

WILKINSON J.H., [1965], *"The Algebraic Eigenvalue Problem"*,
Oxford University Press.

WULF W.A. and BELL C.G., [1972], *"C.mmp, a multi-mini-processor"*,
AFIPS Fall 1972, AFIPS Press, Montvale, N.J., Volume 41, Part 2,
pp.765-777.

YOUNG D.M., [1954], *"Iterative Methods for Solving Partial Differential Equations of Elliptic Type"*,
Trans. Amer. Math. Soc., 76, pp.92-111.

BURKE A.W., GOLDSTINE H.H., and VON NEUMANN [1971] *"Preliminary Discussions of the Logical Design of an Electronic Computer"*, Computer Structures: Reading and Examples, Bell and Newell (eds.), pp 92-119.

DENNIS J.B. and MISUNAS D.P. [1974] *"A Computer Architecture for highly Parallel Signal Processing"*, Proc. of the ACM National Conference, New York, pp 402-409.

DENNIS J.B. and MISUNAS D.P. [1975] *"A Preliminary Architecture for a Basic Data Flow Processor"*, Proc. of the second annual Symposium on Computer Architecture, IEEE, pp 126-132.

RUMBAUGH J.E. [1975]. *"A Parallel Asynchonous Computer Architecture for Data Flow Programs"*, MIT Project MAC, TR-150, Mass., USA.

RUMBAUGH J.E. [1977] *"A Data Flow Multiprocessor"*, IEEE Trans. on Comp., Volume C-26, pp. 138-146.

APPENDIX A

An essential part of the analysis of the run time of Parallel Quicksort is the estimation of the frequencies with which each statement is executed. Thus, to complete the analysis, it is necessary to know the time required to execute each statement. This information is also required so that the method can be simulated accurately.

The statement times will of course vary from computer to computer but since the algorithm is not designed for a particular computer, we shall use computer independent timings.

To derive computer independent statement times, we define a matrix T such that $t_{i,j}$ is the time for statement i (i=1,2,...n) when executed on computer j (j=1,2,...,m). Obviously there will not be a constant ratio between statement times on different computers but it is a reasonable assumption to make that,

$$t_{i,j} \simeq s_i \times M_j \quad (i=1,2,...n, \quad j=1,2,...m) \tag{1}$$

where s_i is a time factor dependent on the statement only

and M_j is a time factor depending on the computer only.

If we introduce a factor $R_{i,j}$ into equation (1) where $R_{i,j}$ is close to unity, then we have

$$t_{i,j} = s_i \times M_j \times R_{i,j}, \ (i=1,2,...n, \ j=1,2,...m). \tag{2}$$

Now, in order to calculate the s_i and M_j we may use the method of least squares to minimise errors by first assuming that s_i and M_j are exact and that the errors in $R_{i,j}$ are the amount by which they differ from unity. Then, taking logs, equation (2) becomes,

$$\ell n \ t_{i,j} = \ell n \ s_i + \ell n \ M_j + \ell n \ R_{i,j}, \ (i=1,2,...n, \ j=1,2,...m) \tag{3}$$

where $\ell nX = \log_e(X)$,

and since $\ell n1=0$, we must now minimise $\ell n \ R_{i,j}$. So let E, the sum of the squared errors, be,

$$E = \sum_i \sum_j \{\ell n \ R_{i,j}\}^2 = \sum_i \sum_j \{\ell n \ s_i + \ell n \ M_j - \ell n \ t_{i,j}\}^2. \tag{4}$$

To minimise E with respect to s_i and M_j, equation (4) must be differentiated with respect to these variables giving,

$$\left.\begin{aligned} &\sum_j(\ell n\ s_i + \ell n\ M_j - \ell n\ t_{i,j}) = m\ \ell n\ s_i + \sum_j \ell n\ M_j - \ell n\ t_{i*} = 0 \\ \text{and}\quad &\sum_i(\ell n\ s_i + \ell n\ M_j - \ell n\ t_{i,j}) = \sum_i \ell n\ s_i + n\ell n\ M_j - \ell n\ t_{*j} = 0 \end{aligned}\right\} \quad (5)$$

where $\ell n\ t_{i*}$ and $\ell n t_{*j}$ are row and column sums respectively of the matrix $(\ell n\ t_{i,j})$. By taking $M_1=1$, these equations may be solved explicitly to give,

$$\left.\begin{aligned} &\ell n\ M_j = (\ell n\ t_{*j} - \ell n\ t_{*1})/n \quad \text{for } j=1,2,\ldots m \\ \text{and}\quad &\ell n\ s_i = \ell n\ t_{i*}/m + \ell n\ t_{*1}/n - \ell n\ t_{**}/mn \quad \text{for } i=1,2,\ldots n \end{aligned}\right\} (6)$$

where $\ell n\ t_{**}$ is the sum of all the elements of $(\ell n\ t_{i,j})$.

Finally, by taking exponentials, values of the s_i and M_j can be calculated easily, in particular, the s_i.

Computer independent statement times have been obtained in this manner for Algol 60 by Wichmann [1973], and those of relevance to the analysis of Program 7 are as follows, where the units are approximately one machine instruction time:-

for operators, +,- = 1 unit,

and '/' (integer division) = 6 units.

Access to simple and array variables = 1 unit,

and use of constants = 1 unit.

Array variable access is allowed for by weighting the opening bracket thus, [= 3 units,

and commas separating subscripts as , = 3 units.

The assignment symbol ← is ignored.

For boolean operators we have 'GT','LT' = 2 units,

and for conditional statements 'IF' = 0 units,

'THEN' = 2 units,

and 'ELSE' = 1 unit.

Loop statements are weighted according to the for list elements which are,

(i) <a> 'WHILE' <b> where for each time around the loop allow (expression <a> + expression <b> + 8) units and the same amount for the final test,

(ii) <a> 'STEP' <b> 'UNTIL' <c> where for the initial assignment and test allow (expression <a> + expression <c> + 3) units and for each time round the loop (expression <b> + expression <c> + 12) units.

For the branch statement 'GOTO' = 2 units,

and for entry and exit to a block allow 10 units.

'BEGIN' and 'END' of compound statements and ; are ignored.

Procedure calls are allowed for as follows:

	procedure identifier	25 units,
	parameter bracket (	12 units,
	, separating parameters	8 units,
	integers by value	1 unit,
and	array identifiers by name	1 unit.

Fork and Join statements are difficult to assess since they have rarely been implemented. However, their use in program 7 is more than allowed for by the procedure calls.

In the remainder of this appendix we shall derive proofs for identities used in the analysis of the run time of the Parallel Quicksort Method. First we shall list the notations used:

$$\sum_{j=1}^{n} a_j = a_1 + a_2 + \ldots a_n \quad , \tag{7}$$

$$\prod_{1 \leq j \leq n} a_j = a_1 \times a_2 \times \ldots \times a_n \quad , \tag{8}$$

$$H_n = \sum_{j=1}^{n} \frac{1}{j} \quad (n^{th} \text{ harmonic number}) \quad , \tag{9}$$

$$n! = \prod_{1 \leq j \leq n} j \quad (n \text{ factorial}) \quad , \tag{10}$$

and $\Delta f(n) = f(n+1) - f(n)$ (forward difference operator) . (11)

Now let us consider the sum of the first n harmonic numbers. Clearly we have,

$$\sum_{k=1}^{n} H_k = \sum_{k=1}^{n} \sum_{j=1}^{k} \frac{1}{j} \quad ,$$

and by interchanging the order of summation we obtain,

$$\begin{aligned} \sum_{k=1}^{n} H_k &= \sum_{j=1}^{n} \sum_{k=j}^{n} \frac{1}{j} \\ &= \sum_{j=1}^{n} \frac{1}{j} \sum_{k=j}^{n} 1 \\ &= \sum_{j=1}^{n} \frac{n-j+1}{j} \\ &= (n+1) \sum_{j=1}^{n} \frac{1}{j} - \sum_{j=1}^{n} 1 \\ &= (n+1)H_n - n \quad , \end{aligned}$$

which leads to,

$$\sum_{k=1}^{n} H_k = (n+1)(H_{n+1}-1) \quad . \tag{12}$$

Next we must consider binomial coefficients $\binom{n}{k}$ defined as

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \qquad n \geq k \geq 0$$

or in its less restrictive form,

$$\binom{n}{k} = \begin{cases} \frac{1}{k!} \prod_{j=1}^{k} (n-k+j) & \text{for integer } k \geq 0 \ , \\ 0 & \text{for integer } k<0. \end{cases}$$

In particular, we observe that,

$$\binom{n}{0} = 1 \ , \quad \binom{n}{1} = n \quad \text{and} \quad \binom{n}{k} = \binom{n}{n-k} \text{ for positive integers.}$$

A not so obvious relationship is

$$\binom{-n}{k} = (-1)^k \binom{n+k-1}{k} \quad , \tag{13}$$

which may be proved as follows,

$$\binom{-n}{k} = \frac{1}{k!} \prod_{j=1}^{k} (-n-k+j)$$

$$= \frac{(-1)^k}{k!} \prod_{j=1}^{k} (n+k-j)$$

$$= (-1)^k \frac{(n+k-1)!}{(n-1)!k!}$$

$$= (-1)^k \binom{n+k-1}{k} \quad .$$

The addition formula,

$$\binom{n}{k} + \binom{n}{k+1} = \binom{n+1}{k+1} \quad , \tag{14}$$

is easily proved as follows:

$$\binom{n+1}{k+1} - \binom{n}{k+1} = \frac{1}{(k+1)!} \left(\prod_{j=1}^{k+1} (n-k+j) - \prod_{j=1}^{k+1} (n-k-1+j) \right)$$

$$= \frac{1}{(k+1)!} \left((n+1) \prod_{j=1}^{k} (n-k+j) - (n-k) \prod_{j=1}^{k} (n-k+j) \right)$$

$$= \frac{1}{k!} \prod_{j=1}^{k} (n-k+j)$$

$$= \binom{n}{k} \quad ,$$

which may be rearranged to give formula (14).

The relationship,

$$\sum_{k=0}^{n} \binom{k}{m} = \binom{n+1}{m+1} \quad \text{integers } n,m \geq 0 \tag{15}$$

may be proved by induction.

Obviously, letting n=1, we have

$$\sum_{k=0}^{1} \binom{k}{m} = \binom{0}{m} + \binom{1}{m}$$

and

$$\binom{n+1}{m+1} = \binom{2}{m+1}$$

If we generate these quantities for different values of m we find that they are equal. Considering $\sum_{k=0}^{n+1} \binom{k}{m}$ we find that,

$$\sum_{k=0}^{n+1} \binom{k}{m} = \sum_{k=0}^{n} \binom{k}{m} + \binom{n+1}{m}$$

$$= \binom{n+1}{m+1} + \binom{n+1}{m}$$

$$= \binom{n+2}{m+1} \quad .$$

Thus, equation (15) is true for n=1 and if it is true for n, it is true for n+1, so by induction it is true for all n.

Another important formula is

$$\sum_{k=1}^{n} \binom{k}{m} H_k = \binom{n+1}{m+1}\left(H_{n+1} - \frac{1}{m+1}\right) \tag{16}$$

which may be proved as follows:

$$\sum_{k=1}^{n} \binom{k}{m} H_k = \sum_{k=1}^{k} \binom{k}{m} \sum_{j=1}^{k} \frac{1}{j} \quad ,$$

and by interchanging the order of summation we have,

$$\sum_{j=1}^{n} \frac{1}{j} \sum_{k=j}^{n} \binom{k}{m} = \sum_{j=1}^{n} \frac{1}{j} \sum_{k=0}^{n} \binom{k}{m} - \sum_{j=1}^{n} \frac{1}{j} \sum_{k=0}^{j-1} \binom{k}{m}$$

$$= \sum_{j=1}^{n} \frac{1}{j} \binom{n+1}{m+1} - \sum_{j=1}^{n} \frac{1}{j} \binom{j}{m+1}$$

$$= \binom{n+1}{m+1} H_n - \frac{1}{m+1} \sum_{j=1}^{n} \binom{j-1}{m}$$

$$= \binom{n+1}{m+1} H_n - \frac{1}{m+1} \binom{n}{m+1}$$

which finally leads to the result

$$\sum_{k=1}^{n} \binom{k}{m} H_k = \binom{n+1}{m+1}\left(H_{n+1} - \frac{1}{m+1}\right) \quad .$$

To obtain the remaining results necessary for the run time analysis we must consider generating functions. A generating function A(z) defined as

$$A(z) = \sum_{k \geq 0} a_k z^k \quad , \tag{17}$$

is the generating function for the sequence $\langle a_k \rangle$, e.g., the generating function for binomial coefficients is

$$A(z) = (1+z)^n = \sum_{k \geq 0} \binom{n}{k} z^k \quad . \tag{18}$$

If result (13) is substituted into equation (18) we have,

$$(1+z)^n = \sum_{k \geq 0} (-1)^k \binom{-n+k-1}{k} z^k$$

or

$$\frac{1}{(1-z)^{n+1}} = \sum_{k \geq 0} \binom{n+k}{k} z^k$$

$$= \sum_{k\geq n} \binom{k}{n} z^{k-n}$$

which leads to,

$$\frac{z^n}{(1-z)^{n+1}} = \sum_{k\geq 0} \binom{k}{n} z^k \quad . \tag{19}$$

In particular, we have

$$\frac{1}{(1-z)} = \sum_{k\geq 0} z^k \quad \text{and} \quad \frac{z}{(1-z)^2} = \sum_{k\geq 0} kz^k \quad .$$

Now by considering the quantity $(1+z)^r(1+z)^s$ we have

$$(1+z)^r(1+z)^s = \sum_{k\geq 0}\binom{r}{k}z^k \sum_{m\geq 0}\binom{s}{m}z^m$$

$$= \sum_{k\geq 0}\binom{r}{k} \sum_{m\geq k}\binom{s}{m-k}z^m$$

$$= \sum_{m\geq 0} \sum_{k=0}^{m} \binom{r}{k}\binom{s}{m-k}z^m \quad ,$$

but

$$(1+z)^r(1+z)^s = (1+z)^{r+s}$$

$$= \sum_{m\geq 0}\binom{r+s}{m}z^m \quad .$$

If these infinite series are equal, the coefficients of z^m must be equal and therefore,

$$\sum_{k=0}^{m}\binom{r}{k}\binom{s}{m-k} = \binom{r+s}{m} \quad . \tag{20}$$

We already know that $\binom{r}{k}=\binom{r}{r-k}=(-1)^{r-k}\binom{-k-1}{r-k}$ when r is an integer, and so another important identity may be obtained by substituting these values into equation (20), which gives,

$$\sum_k (-1)^{r-k}\binom{-k-1}{r-k}(-1)^{s-m+k}\binom{-m+k-1}{s-m+k} = (-1)^{r+s-m}\binom{-m-1}{r+s-m} \quad .$$

If we now replace k by k-n-1, where n is an integer, and cancel (-1) factors we have,

$$\sum_k \binom{n-k}{r+n+1-k}\binom{-m-n-2+k}{s-m-n-1+k} = \binom{-m-1}{r+s-m} \quad ,$$

and changing variables to m=-m-n-2, r=-r-1 and s=-s-1 leads to

$$\sum_k \binom{n-k}{n-r-k}\binom{m+k}{m-s+k} = \binom{m+n+1}{r+s+1} \quad .$$

Finally, if $0\leq n-r-k\leq n-k$ and $0\leq m-s+k\leq m+k$, then we have

$$\sum_{k=0}^{n} \binom{n-k}{r}\binom{m+k}{s} = \binom{m+n+1}{r+s+1} \qquad (21)$$

for integers $n \geq s \geq 0$, $m, r \geq 0$.

Now returning to generating functions, if we let A(z) be the generating function for $\langle a_z \rangle$ and B(z) be the generating function for $\langle b_z \rangle$ then,

$$\begin{aligned} A(z)B(z) &= \sum_{j\geq 0} a_j z^j \cdot \sum_{k\geq 0} b_k z^k \\ &= \sum_{j\geq 0} a_j \sum_{k\geq j} b_{k-j} z^k \\ &= \sum_{k\geq 0} \sum_{j=0}^{k} a_j b_{k-j} \cdot z^k \quad , \end{aligned}$$

and thus A(z)B(z) is the generating function for $\langle \sum_{j=0}^{k} a_j b_{k-j} \rangle$. It is not difficult to see that if $B(z)=\frac{1-z}{z}$ and $a_0=0$ then $\frac{(1-z)}{z}A(z)$ is the generating function for $\langle \Delta a_k \rangle$, i.e.,

$$\frac{1-z}{z}A(z) = \sum_{k\geq 0} \Delta a_k z^k \quad \text{when } A(z) = \sum_{k\geq 1} a_k z^k \ . \qquad (22)$$

This final equation completes the proofs of the identities that are required in Chapter 5 for the analysis of the run time of the Parallel Quicksort Algorithm.

APPENDIX B

In this Appendix, programs 2,4,8 and 9 use the procedure

FO1ARA(L,S,U,A1,A2,A[K],B[K],K,A3,A4)

which is a NAG library routine (NAG, 1976) that accumulates the inner product,

$$A3 = A1 + \sum_{K=L}^{U} A_K . B_K \quad ,$$

to double precision and rounds the result to single precision.

PROGRAM 1

```
'PRØCEDURE' FØLTRID1(N,A,B,C,D);
'CØMMENT' Procedure solves the set of linear equations Ax=d, where A is
    an (N×N) tridiagonal matrix, using the parallel factorisation method
    without partial pivoting (see Chapter 3). The main diagonal of
    matrix A is stored in vector A, the lower sub-diagonal in vector B
    and the upper sub-diagonal in vector C. On input, vector D contains
    the right hand side of the system of equations, during computation
    the intermediate solution and on exit it contains the solution x.
    Matrices P and Q are overwritten on A,B and C. ;
'ARRAY' A,B,C,D;'INTEGER' N;
'BEGIN'
    'INTEGER' S,I,J;
    S←(N+1)'/'2;
    'CØMMENT' The factorisation process. ;
    'FØRK' L1,L2;

L1: B[2]←B[2]/A[1];
    'FØR' I←2 'STEP' 1 'UNTIL' S-1 'DØ'
    'BEGIN'
        A[I]←A[I]-B[I]*C[I];
        B[I+1]←B[I+1]/A[I]
    'END';
    'GØTØ' L3;

L2: C[N]←C[N]/A[N];
    'FØR' J←N-1 'STEP' 1 'UNTIL' S+1 'DØ'
    'BEGIN'
        A[J]←A[J]-C[J+1]*B[J+1];
        C[J]←C[J]/A[J]
    'END';
    'GØTØ' L3;

L3: 'JØIN' L1,L2;
    A[S]←A[S]-(B[S]*C[S]+C[S+1]*B[S+1]);
    'CØMMENT' The inward substitution process. ;
    'FØRK' L4,L5;

L4: 'FØR' I←2 'STEP' 1 'UNTIL' S-1 'DØ' D[I]←D[I]-B[I]*D[I-1];
    'GØTØ' L6;

L5: 'FØR' J←N-1 'STEP' -1 'UNTIL' S+1 'DØ' D[J]←D[J]-C[J+1]*D[J+1];
    'GØTØ' L6;

L6: 'JØIN' L4,L5;
    D[S]←D[S]-(B[S]*D[S-1]+C[S+1]*D[S+1]);
    'CØMMENT' The outward substitution process. ;
    D[S]←D[S]/A[S];
    'FØRK' L7,L8;

L7: 'FØR' I←S-1 'STEP' -1 'UNTIL' 1 'DØ' D[I]←(D[I]-C[I+1]*D[I+1])/A[I];
    'GØTØ' L9;

L8: 'FØR' J←S+1 'STEP' 1 'UNTIL' N 'DØ' D[J]←(D[J]-B[J]*D[J-1])/A[J];
    'GØTØ' L9;

L9: 'JØIN' L7,L8
'END';
```

PROGRAM 2

```
'PRØCEDURE' FØLTRID2(N,A,D);
'CØMMENT' Procedure solves the set of linear equations Ax=d, where A is
    an (N×N) tridiagonal matrix, using the parallel factorisation method
    with partial pivoting (Chapter 3). On input, vector D holds the
    right hand side of the system of equations, during computation the
    intermediate solution and, on exit, the final solution x. Matrices
    P and Q are overwritten on A. ;
'ARRAY' A,D;'INTEGER'N;
'BEGIN'
    'INTEGER' S,I,J,K1,K2,L1,L2,U1,U2;
    'REAL' W1,W2,E,A2,A4;
    'ARRAY' R[1:N];
    S←(N+1)'/'2;
    'CØMMENT' The factorisation process. ;
    'FØRK' L1,L2;

L1: 'FØR' I←1 'STEP' 1 'UNTIL' S-1 'DØ'
    'BEGIN'
        U1←'IF' I 'GT' 3 'THEN' I-2 'ELSE' 1;
        'FØR' K1←0,1 'DØ'
        'BEGIN'
           FOlARA(U1,1,I-1,-A[I+K1,I],A2,A[I+K1,L1],A[L1,I],L1,W1,A4);
           R[I+K1]←-W1
        'END';
        'IF' ABS(R[I+1])'GT' ABS(R[I])'THEN'
        'BEGIN'
           'FØR' K1←1 'STEP' 1 'UNTIL' I+2 'DØ'
           'BEGIN'
              W1←A[I,K1];A[I,K1]←A[I+1,K1];A[I+1,K1]←W1
           'END';
           W1←D[I];D[I]←D[I+1];D[I+1]←W1;
           W1←R[I];R[I]←R[I+1];R[I+1]←W1
        'END';
        A[I,I]←R[I];
        'FØR' K1←1,2 'DØ'
        'BEGIN'
           FOlARA(U1,1,I-1,-A[I,I+K1],A2,A[I,L1],A[L1,I+K1],L1,W1,A4);
           A[I,I+K1]←-W1
        'END';
        A[I+1,I]←R[I+1]/A[I,I]
    'END';
    'GØTØ' L3;

L2: 'FØR' J←N 'STEP' -1 'UNTIL' S+2 'DØ'
    'BEGIN'
        U2←'IF' N-J+1 'GT' 3 'THEN' J+2 'ELSE' N;
        'FØR K2←0,1 'DØ'
        'BEGIN'
           FOlARA(J+1,1,U2,-A[J-K2,J],A2,A[J-K2,L2],A[L2,J],L2,W2,A4);
           R[J-K2]←-W2
        'END';
        'IF' ABS(R[J-1])'GT' ABS(R[J])'THEN'
        'BEGIN'
           'FØR' K2←N 'STEP' -1 'UNTIL' J-2 'DØ'
           'BEGIN'
              W2←A[J,K2];A[J,K2]←A[J-1,K2];A[J-1,K2]←W2
           'END';
```

```
            W2←D[J];D[J]←D[J-1];D[J-1]←W2;
            W2←R[J];R[J]←R[J-1];R[J-1]←W2
        'END';
        A[J,J]←R[J];
        'FØR' K←1,2 'DØ'
        'BEGIN'
            FO1ARA(J+1,1,U2,-A[J,J-K2],A2,A[J,L2],A[L2,J-K2],L2,W2,A4);
            A[J,J-K2]←-W2
        'END';
        A[J-1,J]←R[J-1]/A[J,J]
    'END';
    'GØTØ' L3;

L3: 'JØIN' L1,L2;
    'FØR' K2←0,1 'DØ'
    'BEGIN'
        FO1ARA(S+2,1,S+3,-A[S-K2+1,S+1],A2,A[S-K2+1,L2],A[L2,S+1],L2,E,A4);
        FO1ARA(S-1,1,S-1,E,A2,A[S-K2+1,L2],A[L2,S+1],L2,W2,A4);
        R[S-K2+1]←-W2
    'END';
    'IF' ABS(R[S])'GT' ABS(R[S+1])'THEN'
    'BEGIN'
        'FØR' K2←1 'STEP' 1 'UNTIL' N 'DØ'
        'BEGIN'
            W1←A[S+1,K2];A[S+1,K2]←A[S,K2];A[S,K2]←W2
        'END';
        W2←D[S+1];D[S+1]←D[S];D[S]←W2;
        W2←R[S+1];R[S+1]←R[S];R[S]←W2
    'END';
    A[S+1,S+1]←R[S+1];
    FO1ARA(S+2,1,S+3,-A[S+1,S],A2,A[S+1,L2],A[L2,S],L2,E,A4);
    FO1ARA(S-2,1,S-1,E,A2,A[S+1,L2],A[L2,S],L2,W2,A4);
    A[S+1,S]←-W2;
    A[S,S+1]←R[S]/A[S+1,S+1];
    FO1ARA(1,1,2,-A[S,S]A2,A[S,S-L2],A[S-L2,S],L2,E,A4);
    FO1ARA(1,1,2,E,A2,A[S,S+L2],A[S+L2,S],L2,W2,A4);
    A[S,S]←-W2;
    'CØMMENT' The inward substitution process. ;
    'FØRK' L4,L5;

L4: 'FØR' I←2 'STEP' 1 'UNTIL' S-1 'DØ'
    'BEGIN'
       FO1ARA(1,1,I-1,-D[I],A2,A[I,L1],D[L1],L1,W1,A4);
       D[I]←- W1
    'END';
    'GØTØ' L6;

L5: 'FØR' J←N-1 'STEP' -1 ' UNTIL' S+2 'DØ'
    'BEGIN'
        FO1ARA(J+1,1,N,-D[J],A2,A[J,L2],D[L2],L2,W2,A4);
        D[J]←-W2
    'END';
    'GØTØ' L6;

L6: 'JØIN' L4,L5;
    'FØR' J←S+1,S 'DØ'
    'BEGIN'
        FO1ARA(J+1,1,N,-D[J],A2,A[J,L2],D[L2],L2,E,A4);
        FO1ARA(1,1,S-1,E,A2,A[J,L2],D[L2],L2,W2,A4);
        D[J]←-W2;
    'END';
```

```
    'CØMMENT' The outward substitution process. ;
    D[S]←D[S]/A[S,S];
    D[S+1]←(D[S]-A[S+1,S]*D[S])/A[S+1,S+1];
    'FØRK' L7,L8;

L7: 'FØR' J←S+2 'STEP' 1 'UNTIL' N 'DØ'
    'BEGIN'
        FO1ARA(J-2,1,J-1,-D[J],A2,A[J,K2],D[K2],K2,W2,A4);
        D[J]←-W2/A[J,J]
    'END';
    'GØTØ' L9;

L8: 'FØR' I←S-1 'STEP' -1 'UNTIL' 1 'DØ'
    'BEGIN'
        FO1ARA(I+1,1,I+2,-D[I],A2,A[I,L1],D[L1],L1,W1,A4);
        D[I]←-W1/A[I,I]
    'END';
    'GØTØ' L9;

L9: 'JØIN' L7,L8
'END';
```

PROGRAM 3

```
'PRØCEDURE' FØLTRID3(N,A,D,M);
'CØMMENT' Procedure solves the set of linear equations Ax=d, where A is
    an (N×N) banded matrix of semi-bandwidth M, using the parallel
    factorisation method without partial pivoting (Chapter 3). Matrix A
    is stored in an (N×(2M-1)) array A, the main diagonal being held in
    column M, the lower sub-diagonals in columns 1 to M-1 and the upper
    sub-diagonals in columns M+1 to 2M-1. On input, vector D contains
    the right hand side of the system of equations, during computation
    the intermediate solution and on exit the final solution x. Matrices
    P and Q are overwritten on A.  ;
'ARRAY' A,B; 'INTEGER' M,N;
'BEGIN'
    'INTEGER' S,I,J,K1,K2,L1,L2,U1,U2;
    S←(N-M+3)'/'2;
    'CØMMENT' The factorisation process.  ;
    'FØRK' L1,L2;

L1: 'FØR' I←1 'STEP' 1 'UNTIL' S-1 'DØ'
    'BEGIN'
        'FØR' K1←0 'STEP' 1 'UNTIL' M-1 'DØ'
        'BEGIN'
            U1←'IF' M-K1 'GT' I 'THEN' I-1 'ELSE' M-K1-1;
            'FØR' L1←1'STEP' 1 'UNTIL' U1 'DØ' A[I+K1,M+K1]←A[I+K1,M+K1]
            -A[I,M-L1]*A[I+K1,M+K1+L1]
        'END';
        'FØR' K1←1 'STEP' 1 'UNTIL' M-1 'DØ'
        'BEGIN'
            U1←'IF' M-K1 'GT' I 'THEN' I-1 'ELSE' M-K1-1;
            'FØR' L1←1 'STEP' 1 'UNTIL' U1 'DØ' A[I+K1,M-K1]←A[I+K1,M-K1]
            -A[I+K1,M-K1-L1]*A[I,M+L1];
            A[I+K1,M-K1]←A[I+K1,M-K1]/A[I,M]
        'END'
    'END';
    'GØTØ' L3;

L2: 'FØR' J←N 'STEP' -1 'UNTIL' S+M-1 'DØ'
    'BEGIN'
        'FØR' K2←0 'STEP' 1 'UNTIL' M-1 'DØ'
        'BEGIN'
            U2←'IF' M-K2 'GT' N-J+1 'THEN' N-J 'ELSE' M-K2-1;
            'FOR' L2←1 'STEP' 1 'UNTIL' U2 'DO' A[J,M-K2]←A[J,M-K2]
            -A[J+L2,M+L2]*A[J+L2,M-K2-L2]
        'END';
        'FØR' K2←1 'STEP' 1 'UNTIL' M-1 'DØ'
        'BEGIN'
            U2←'M-K2 'GT' N-J+1 'THEN' N-J 'ELSE' M-K2-1;
            'FØR' L2←1 'STEP' 1 'UNTIL' U2 'DØ' A[J,M+K2]←A[J,M+K2]
            -A[J+L2,M+K2+L2]*A[J+L2,M-L2];
            A[J,M+K2]←A[J,M+K2]/A[J,M]
        'END'
    'END';
    'GØTØ' L3;

L3: 'JØIN' L1,L2;
    'FØR' J←S+M-2 'STEP' -1 'UNTIL' S 'DØ'
    'BEGIN'
```

```
        'FØR' K2←0 'STEP' 1 'UNTIL' J-S 'DØ'
        'BEGIN'
            'FØR' L2←1 'STEP' 1 'UNTIL' M-K2-1 'DØ' A[J,M-K2]←A[J,M-K2]
            -A[J+L2,M+L2]*A[J+L2,M-K2+L2];
            'FØR' L2←M-1 'STEP' -1 'UNTIL' J-S+1 'DØ' A[J,M-K2]←A[J,M-K2]
            -A[J,M-L2]*A[J-K2,M-K2+L2]
        'END';
        'FØR' K2←1 'STEP' 1 'UNTIL' J-S 'DØ'
        'BEGIN'
            'FØR' L2←1 'STEP' 1 'UNTIL' M-K2-1 'DØ' A[J,M+K2]←A[J,M+K2]
            -A[J+L2,M+K2+L2]*A[J+L2,M-K2];
            'FOR' L2←M-1 'STEP' -1 'UNTIL' J-S+1 'DØ' A[J,M+K2]←A[J,M+K2]
            -A[J-K2,M+K2-L2]*A[J,M+L2];
            A[J,M+K2]←A[J,M+K2]/A[J,M]
        'END'
    'END';
    'CØMMENT' The inward substitution process.  ;
    'FØRK' L4,L5;

L4: 'FØR' I←2 'STEP' 1 'UNTIL' S-1 'DØ'
    'BEGIN'
        U1←'IF' M 'GT' I 'THEN' I-1 'ELSE' M-1;
        'FØR' L1←1 'STEP' 1 'UNTIL' U1 'DØ' D[I]←D[I]-A[I,M-L1]*D[I-L1]
    'END';
    'GØTØ' L6;

L5: 'FØR' J←N-1 'STEP' -1 'UNTIL' S+M-1 'DØ'
    'BEGIN'
        U2←'IF' M 'GT' N-J+1 'THEN' N-J 'ELSE' M-1;
        'FOR' L2←1 'STEP' 1 'UNTIL' U2 'DO' D[J]←D[J]-A[J+L2,M+L2]*D[J+L2]
    'END';
    'GØTØ' L6;

L6: 'JØIN' L4,L5;
    'FØR' J←S+M-2 'STEP' -1 'UNTIL' S 'DØ'
    'BEGIN'
        'FØR' L2←1 'STEP' 1 'UNTIL' M-1 'DØ' D[J]←D[J]-A[J+L2,M+L2]*D[J+L2];
        'FØR' L2←M-1 'STEP' -1 'UNTIL' J-S+1 'DØ' D[J]←D[J]-A[J,M-L2]*D[J-L2]
    'END';
    'CØMMENT' The outward substitution process. ;
    D[S]←D[S]/A[S,M];
    'FØR' J←S+1 'STEP' 1 'UNTIL' S+M-2 'DØ'
    'BEGIN'
        'FØR K2←1 'STEP' 1 'UNTIL' J-S 'DØ' D[J]←D[J]-A[J,M-K2]*D[J-K2];
        D[J]←D[J]/A[J,M]
    'END';
    'FØRK' L7,L8;

L7: 'FØR' J←S+M-1 'STEP' 1 'UNTIL' N 'DØ'
    'BEGIN'
        'FØR' K2←1 'STEP' 1 'UNTIL' M-1 'DO' D[J]←D[J]-A[J,M-K2]*D[J-K2];
        D[J]←D[J]/A[J,M]
    'END';
    'GØTØ' L9;

L8: 'FØR' I←S-1 'STEP' -1 'UNTIL' 1 'DØ'
    'BEGIN'
        'FOR' K1←1 'STEP' 1 'UNTIL' M-1 'DØ' D[I]←D[I]-A[I+K1,M+K1]*D[I+K1];
        D[I]←D[I]/A[I,M]
    'END';
    'GØTØ' L9;

L9: 'JØIN' L7,L8
'END';
```

PROGRAM 4

```
'PRØCEDURE' FØLTRID4(N,A,D,M);
'CØMMENT' Procedure solves the set of linear equations Ax=d, where A
    is an (N×N) banded matrix of semi-bandwidth M, using the parallel
    factorisation method with partial pivoting (Chapter 3). On input,
    matrix D holds the right hand side of the system of equations,
    during computation the intermediate solution and on exit the final
    solution x. The matrices P and Q are overwritten on A.
'ARRAY'A,D; 'INTEGER' M,N;
'BEGIN'
    'INTEGER'S,I,J,K1,K2,L1,L2,U1,U2,MAX1,MAX2;
    'REAL' W1,W2,E,A2,A4;
    'ARRAY' R[1:N];
    S←(N+5)'/'2-M;
    'CØMMENT' The factorisation process. ;
    'FØRK' L1,L2;

L1: 'FØR' I←1 'STEP' 1 'UNTIL' S-1 'DØ'
    'BEGIN'
        U1←'IF' I 'GT' 2*M-1 'THEN' I-2*(M-1)'ELSE' 1;
        'FØR' K1←O 'STEP' 1 'UNTIL' M-1 'DØ'
        'BEGIN'
            FO1ARA(U1,1,I-1,O,A2,A[I+K1,L1],A[L1,I],L1,R[I+K1],A4);
            R[I+K1]←A[I+K1,I]-R[I+K1]
        'END';
        MAX1←I;
        'FØR' K1←1 'STEP' 1 'UNTIL' M-1 'DØ'
            'IF' ABS(R[I+K1])'GT'ABS(R[MAX1])'THEN' MAX1←I+K1;
        'IF' MAX1 'NE' I 'THEN'
        'BEGIN'
            'FØR' K1←1 'STEP' 1 'UNTIL' I+2*(M-1)'DØ'
            'BEGIN'
                W1←A[I,K1];A[I,K1]←A[MAX1,K1];A[MAX1,K1]←W1
            'END';
            W1←D[I];D[I]←D[MAX1];D[MAX1]←W1;
            W1←R[I];R[I]←R[MAX1];R[MAX1]←W1
        'END';
        A[I,I]←R[I];
        'FØR' K1←1 'STEP' 1 'UNTIL' 2*(M-1) 'DØ'
        'BEGIN'
            FO1ARA(U1,1,I-1,O,A2,A[I,L1],A[L1,I+K1],L,W,A4);
            A[I,I+K1]←A[I,I+K1]-W
        'END';
        'FØR' K1←1 'STEP' 1 'UNTIL' M-1 'DØ' A[I+K1,I]←R[I+K1]/A[I,I]
    'END';
    'GØTØ' L3;

L2: 'FØR' J←N 'STEP' -1 'UNTIL' S+2*(M-1)'DØ'
    'BEGIN'
        U2←'IF' N-J+1 'GT' 2*M-1 'THEN' J+2*(M-1)'ELSE'N;
        'FØR' K2←O 'STEP' 1 'UNTIL' M-1 'DØ'
        'BEGIN'
            FO1ARA(J+1,1,U2,O,A2,A[J-K2,L2],A[L2,J2],L2,R[J-K2],A4);
            R[J-K2]←A[J-K2,J]-R[J-K2]
        'END';
        MAX2←J;
        'FØR' K2←1 'STEP' 1 'UNTIL' M-1 'DØ'
```

```
            'IF' ABS(R[J-K2])'GT'ABS(R[MAX2])'THEN'MAX2←J-K2;
        'IF' MAX2 'NE' J 'THEN'
        'BEGIN'
            'FØR' K2←N 'STEP' -1 'UNTIL' J-2*(M-1)'DØ'
            'BEGIN'
                W2←A[J,K2];A[J,K2]←A[MAX2,K2];A[MAX2,K2]←W2
            'END';
        W2←D[J];D[J]←D[MAX2];D[MAX2]←W2;
        W2←R[J];R[J]←R[MAX2];R[MAX2]←W2;
    'END';
    A[J,J]←R[J];
    'FØR' K2←1 'STEP' 1 'UNTIL' 2*(M-1) 'DØ'
    'BEGIN'
        FO1ARA(J+1,1,U2,0,A2,A[J,L2],A[L2,J-K2],L2,W2,A4);
        A[J,J-K2]←A[J,J-K2]-W2
    'END';
    'FØR' K2←1 'STEP' 1 'UNTIL' M-1 'DØ' A[J-K2,J]←R[J-K2]/A[J,J]
'END';
'GØTØ' L3;

L3: 'JØIN' L1,L2;
    'FØR' J←S+2*M-3 'STEP' -1 'UNTIL' S+1 'DØ'
    'BEGIN'
        U2←'IF' N-J+1 'GT' 2*M-1 'THEN' J+2*(M-1)'ELSE' N;
        'FØR' K2←0 'STEP' 1 'UNTIL' J-S 'DØ'
        'BEGIN'
            FO1ARA(J+1,1,U2,0,A2,A[J-K2,L2],A[L2,J],L2,E,A4);
            FO1ARA(J-2*M+2,1,S-1,E,A2,A[J-K2,L2],A[L2,J],L2,R[J-K2],A4);
            R[J-K2]←A[J-K2,J]-R[J-K2]
        'END';
        MAX2←J;
        'FØR' K2←1 'STEP' 1 'UNTIL' J-S 'DØ'
            'IF' ABS(R[J-K2])'GT'ABS(R[MAX2])'THEN' MAX2←J-K2;
        'IF' MAX2 'NE' J 'THEN'
        'BEGIN'
            'FØR' K2←1 'STEP' 1 'UNTIL' N 'DØ'
            'BEGIN'
                W2←A[J,K2];A[J,K2]←A[MAX2,K2];A[MAX2,K2]←W2
            'END';
            W2←D[J];D[J]←D[MAX2];D[MAX2]←W2;
            W2←R[J];R[J]←R[MAX2];R[MAX2]←W2
        'END';
        A[J,J]←R[J];
        'FØR' K2←1 'STEP' 1 'UNTIL' J-S 'DØ'
        'BEGIN'
            FO1ARA(J+1,1,U2,0,A2,A[J,L2],A[L2,J-K2],L2,E,A4);
            FO1ARA(J-K2-2*M+2,1,S-1,E,A2,A[J,L2],A[L2,J-K2],L2,W2,A4);
            A[J,J-K2]←A[J,J-K2]-W2
        'END';
        'FØR' K2←1 'STEP' 1 'UNTIL' J-S 'DØ' A[J-K2,J]←R[J-K2]/A[J,J]
    'END';
    FO1ARA(1,1,2*M-2,0,A2,A[S,S-L2],A[S-L2,S],L2,E,A4);
    FO1ARA(1,1,2*M-2,E,A2,A[S,S+L2],A[S+L2,S],L2,W2,A4);
    A[S,S]←A[S,S]-W;
    'CØMMENT' Inward substitution process. ;
    'FØRK' L4,L5;
```

```
L4: 'FØR' I←2 'STEP' 1 'UNTIL' S-1 'DØ'
    'BEGIN'
        FO1ARA(1,1,I-1,O,A2,A[I,L1],D[L1],L1,W1,A4);
        D[I]←D[I]-W1
    'END';
    'GØTØ' L6;

L5: 'FØR' J←N-1 'STEP' -1 'UNTIL' S+2*(M-1)'DØ'
    'BEGIN'
        FO1ARA(J+1,1,N,O,A2,A[J,L2],D[L2],L2,W2,A4);
        D[J]←D[J]-W2
    'END';
    'GØTØ' L6;

L6: 'JØIN' L4,L5;
    'FØR' J←S+2*M-3 'STEP' -1 'UNTIL' S 'DØ'
    'BEGIN'
        FO1ARA(J+1,1,N,O,A2,A[J,L2],D[L2],L2,E,A4);
        FO1ARA(1,1,S-1,E,A2,A[J,L2],D[L2],L2,W2,A4);
        D[J]←D[J]-W2
    'END';
    'CØMMENT' The outward substitution process.  ;
    D[S]←D[S]/A[S,S];
    'FØR' J←S+1 'STEP' 1 'UNTIL' S+2*M-3 'DØ'
    'BEGIN'
        FO1ARA(S,1,J-1,O,A2,A[J,K2],D[K2],K2,W2,A4);
        D[J]←(D[J]-W2)/A[J,J]
    'END';
    'FØRK' L7,L8;

L7: 'FØR' J←S+2*M-2 'STEP' 1 'UNTIL' N 'DØ'
    'BEGIN'
        FO1ARA(J-2*(M-1),1,J-1,O,A2,A[J,K2],D[K2],K2,W2,A4);
        D[J]←(D[J]-W2)/A[J,J]
    'END';
    'GØTØ' L9;

L8: 'FØR'I←S-1 'STEP' -1 'UNTIL' 1 'DØ'
    'BEGIN'
        FO1ARA(I+1,1,I+2*(M-1),O,A2,A[I,L1],D[L1],L1,W1,A4);
        D[I]←(D[I]-W1)/A[I,I]
    'END';
    'GØTØ' L9;

L9: 'JØIN' L7,L8
'END';
```

PROGRAM 5

```
'PRØCEDURE' FØCH1(N,A,B,D);
'CØMMENT' Procedure solves the set of linear equations Ax=d, where A is
    a symmetric tridiagonal (N×N) matrix, using the symmetric parallel
    factorisation method (Chapter 3). The main diagonal of matrix A is
    stored in vector A and the sub-diagonals in vector 3. On input,
    vector D contains the right hand side of the system of equations,
    during computation the intermediate solution and on exit it contains
    the solution x. Matrix P is overwritten on A and B;
'ARRAY' A,B,D;'INTEGER'N;
'BEGIN'
    'INTEGER' S,I,J;
    S←(N+1)'/'2;
    'CØMMENT' The factorisation process. ;
    'FØRK' L1,L2;

L1: A[1]←SQRT(A[1]);B[2]←B[2]/A[1];
    'FØR' I←2 'STEP' 1 'UNTIL' S-1 'DØ'
    'BEGIN'
        A[I]←SQRT(A[I]-B[I]*B[I]);
        B[I+1]←B[I+1]/A[I]
    'END';
    'GØTØ' L3;

L2: A[N]←SQRT(A[N]);B[N]←B[N]/A[N];
    'FØR' J←N-1 'STEP' -1 'UNTIL' S+1 'DØ'
    'BEGIN'
        A[J]←SQRT(A[J]-B[J+1]*B[J+1]);
        B[J]←B[J]/A[J]
    'END';
    'GØTØ' L3;

L3: 'JØIN' L1,L2;
    A[S]←SQRT(A[S]-(B[S]*B[S]+B[S+1]*B[S+1]));
    'CØMMENT' The inward substitution process.  ;
    'FØRK' L4,L5;

L4: D[1]←D[1]/A[1];
    'FØR'  I←2 'STEP' 1 'UNTIL' S-1' 'DØ' D[I]←(D[I]-B[I]*D[I-1])/A[I];
    'GØTØ' L6;

L5: D[N]←D[N]/A[N];
    'FØR' J←N-1 'STEP' -1 'UNTIL' S+1 'DØ' D[J]←(D[J]-B[J+1]*D[J+1])/A[J];
    'GØTØ' L6;

L6: 'JØIN' L4,L5;
    D[S]←(D[S]-(B[S]*D[S-1]+B[S+1]*D[S+1]))/A[S];
    'CØMMENT' The outward substitution process. ;
    D[S]←D[S]/A[S];
    'FØRK' L7,L8;

L7: 'FØR' I←S-1 'STEP' -1 'UNTIL' 1 'DØ' D[I]←(D[I]-B[I+1]*D[I+1])/A[I];
    'GØTØ' L9;

L8: 'FØR' J←S+1 'STEP' 1 'UNTIL' N 'DØ' D[J]←(D[J]-B[J]*D[J-1])/A[J];
    'GØTØ' L9;

L9: 'JØIN' L7,L8
'END';
```

PROGRAM 6

```
'PRØCEDURE' FØCH2(N,A,D,M);
'CØMMENT' Procedure solves the set of linear equations Ax=d, where A is
    an (N×N) symmetric banded matrix of semi-bandwidth M, using the
    symmetric parallel factorisation method (Chapter 3). Matrix A is
    stored in an (N×M) array A, the main diagonal being held in column
    M and the sub-diagonals in columns 1 to M-1. On input, vector D
    contains the right hand side of the system of equations, during
    computation the intermediate solution and on exit the final solution
    x. Matrix P is overwritten on A. ;
'ARRAY' A,D;'INTEGER' N,M;
'BEGIN'
    'INTEGER' S,I,J,K1,K2,L1,L2,U1,U2;
    S←(N-M+3)'/'2;
    'CØMMENT' The factorisation process. ;
    'FØRK' L1,L2;

L1: 'FØR' I←1 'STEP' 1 'UNTIL' S-1 'DØ'
    'BEGIN'
        U1←'IF' M 'GT' I 'THEN' I-1 'ELSE' M-1;
        'FØR' L1←1 'STEP' 1 'UNTIL' U1 'DØ' A[I,M]←A[I,M]-A[I,M-L1]*A[I,M-L1];
        A[I,M]←SQRT(A[I,M]);
        'FØR' K1←1 'STEP' 1 'UNTIL' M-1 'DØ'
        'BEGIN'
            U1←'IF' M-K1 'GT' I 'THEN' I-1 'ELSE' M-K1-1;
            'FØR' L1←1 'STEP' 1 'UNTIL' U1 'DØ' A[I+K1,M-K1]←A[I+K1,M-K1]
            -A[I,M-L1]*A[I+K1,M-K1-L1];
            A[I+K1,M-K1]←A[I+K1,M-K1]/A[I,M]
        'END'
    'END';
    'GØTØ' L3;

L2: 'FØR' J←N 'STEP' -1 'UNTIL' S+M-1 'DØ'
    'BEGIN'
        U2←'IF' M 'GT' N-J+1 'THEN' N-J 'ELSE' M-1;
        'FØR' L2←1 'STEP' 1 'UNTIL' U2 'DØ' A[J,M]←A[J,M]-A[J+L2,M-L2]
        *A[J+L2,M-L2];
        A[J,M]←SQRT(A[J,M]);
        'FØR' K2←1 'STEP' 1 'UNTIL' M-1 'DØ'
        'BEGIN'
            U2←'IF' M-K2 'GT' N-J+1 'THEN' N-J 'ELSE' M-K2-1;
            'FØR' L2←1 'STEP' 1 'UNTIL' U2 'DØ' A[J,M-K2]←A[J,M-K2]
            -A[J+L2,M-L2]*A[J+L2,M-K2-L2];
            A[J,M-K2]←A[J,M-K2]/A[J,M]
        'END'
    'END';
    'GØTØ' L3;

L3: 'JØIN' L1,L2;
    'FØR' J←S+M-2 'STEP' -1 'UNTIL' S 'DØ'
    'BEGIN'
        'FØR' L2←1 'STEP' 1 'UNTIL' M-1 'DØ' A[J,M]←A[J,M]-A[J+L2,M-L2]
        *A[J+L2,M-L2];
        'FØR' L2←J-S+1 'STEP' 1 'UNTIL' M-1 'DØ' A[J,M]←A[J,M]-A[J,M-L2]
        *A[J,M-L2];
        A[J,M]←SQRT(A[J,M]);
        'FØR' K2←1 'STEP' 1 'UNTIL' J-S 'DØ'
        'BEGIN'
```

```
            'FØR' L2←1 'STEP' 1 'UNTIL' M-K2-1 'DØ' A[J,M-K2]←A[J,M-K2]
            -A[J+L2,M-L2]*A[J+L2,M-K2-L2];
            'FOR' L2←J-S+1 'STEP' 1 'UNTIL' M-1 'DO' A[J,M-K2]←A[J,M-K2]
            -A[J,M-L2]*A[J-K2,M+K2-L2];
            A[J,M-K2]←A[J,M-K2]/A[J,M]
        'END'
    'END';
    'CØMMENT' The inward substitition process.  ;
    'FØRK' L4,L5;

L4: 'FØR' I←1 'STEP' 1 'UNTIL' S-1 'DØ'
    'BEGIN'
        U1←'IF' M 'GT' I 'THEN' I-1 'ELSE' M-1;
        'FØR' L1←1 'STEP' 1 'UNTIL' U1 'DØ' D[I]←D[I]-A[I,M-L1]*D[I-L1];
        D[I]←D[I]/A[I,M]
    'END';
    'GØTØ' L6;

L5: 'FØR' J←N 'STEP' -1 'UNTIL' S+M-1 'DØ'
    'BEGIN'
        U2←'IF' M 'GT' N-J+1 'THEN' N-J 'ELSE' M-1;
        'FØR' L2←1 'STEP' 1 'UNTIL' U2 'DØ' D[J]←D[J]-A[J+L2,M-L2]*D[J+L2];
        D[J]←D[J]/A[J,M]
    'END';
    'GØTØ' L6;

L6: 'JØIN' L4,L5;
    'FØR' J←S+M-2 'STEP' -1 'UNTIL' S 'DØ'
    'BEGIN'
        'FØR' L2←1 'STEP' 1 'UNTIL' M-1 'DØ' D[J]←D[J]-A[J+L2,M-L2]*D[J+L2];
        'FØR' L2←J-M+1 'STEP' 1 'UNTIL' M-1 'DØ' D[J]←D[J]-A[J,M-L2]*D[J-L2];
        D[J]←D[J]/A[J,M]
    'END';
    'CØMMENT' The outward substitution process. ;
    D[S]←D[S]/A[S,M];
    'FØR' J←S+1 'STEP' 1 'UNTIL' S+M-2 'DØ'
    'BEGIN'
        'FØR' L2←1 'STEP' 1 'UNTIL' J-S 'DØ' D[J]←D[J]-A[J,M-L2]*D[J-L2];
        D[J]←D[J]/A[J,M]
    'END';
    'FØRK' L7,L8;

L7: 'FØR' J←S+M-1 'STEP' 1 'UNTIL' N 'DØ'
    'BEGIN'
        FØR' L2←1 'STEP' 1 'UNTIL' M-1 'DØ' D[J]←D[J]-A[J,M-L2]*D[J-L2];
        D[J]←D[J]/A[J,M]
    'END';
    'GØTØ' L9;

L8: 'FØR' I←S-1 'STEP' -1 'UNTIL' 1 'DØ'
    'BEGIN'
        'FØR' L1←1 'STEP' 1 'UNTIL' M-1 'DØ' D[I]←D[I]-A[I+L2,M-L2]*A[I+L2];
        D[I]←D[I]/A[I,M]
    'END';
    'GØTØ' L9;

L9: 'JØIN' L7,L8
'END';
```

PROGRAM 7

```
                'PRØCEDURE' PQUICKSØRT(A,L,U);
                'CØMMENT' Procedure sorts N numbers into ascending order using
                    the parallel quicksort method (see Chapter 5). Subsets with
                    more than M elements are sorted using the partitioning
                    procedure and those with less than M,or M elements are sorted
A_N+F_N             using the linear insertion process. ;
                'ARRAY' A;'INTEGER' L,U;'VALUE' L,U;
                'BEGIN'
                    'INTEGER' I,J;'REAL'V,W;
                    'CØMMENT' Test the size of the subset. ;
                    'IF' U-L 'GT' M 'THEN'
                    'BEGIN'
                        'CØMMENT' Select the partition element. ;
A_N                     I←(L+U)'/'2;
A_N                     'IF' A[L] 'GT' A[U] 'THEN'
                        'BEGIN'
A_1                         V←A[L];A[L]←A[U];A[U]←V
                        'END';
A_N                     'IF' A[I] 'GT' A[U] 'THEN'
                        'BEGIN'
A_2                         V←A[I];A[I]←A[U];A[U]←V
                        'END';
A_N                     'IF' A[L] 'GT' A[I] 'THEN'
                        'BEGIN'
A_3                         V←A[L];A[L]←A[I];A[I]←V
                        'END';
                        V←A[I];A[I]←A[L+1];A[L+1]←V;
A_N                     'CØMMENT' Set up pointers and partition on V, the
                            partition element. ;
                        I←L+1; J←U;
C'                  L1: I←I+1;
                        'IF' A[I] 'LT' V 'THEN' 'GØTØ' L1;
C_N-C'              L2: J←J-1;
                        'IF' A[J] 'GT' V 'THEN' 'GØTØ' L2;
                        'CØMMENT' If pointers have crossed, insert partition
                            element, otherwise interchange A[I] and A[J]. ;
A_N+B_N                 'IF' I 'LT' J 'THEN'
                        'BEGIN'
B_N                         W←A[I];A[I]←A[J];A[J]←W;
                            'GØTØ' L1
                        'END';
A_N                     A[L+1]←A[J];A[J]←V;
                        'CØMMENT' Test for largest subset. ;
A_N                     'IF' J-L 'GT' U-J 'THEN'
                        'BEGIN'
                            'CØMMENT' Test smaller subset to see if it has at
                                least 2 elements. ;
A'                          'IF' U 'GT' J+1 'THEN'
                            'BEGIN'
                                'FØRK' L3,L4;

A_4                         L3: PQUICKSØRT(A,L,J-1);
                                'GØTØ' L5;

A_5                         L4: PQUICKSØRT(A,J+1,U);
                                'GØTØ' L5;
```

```
                    L5: 'JØIN' L3,L4;
                    'END'
A6                  'ELSE' PQUICKSØRT(A,L,J-1)
                'END'
                'ELSE'
                'BEGIN'
                    'CØMMENT' Test smaller subset to see if it has at
                        least 2 elements. ;
AN-A'               'IF' J-1 'GT' L 'THEN'
                    'BEGIN'
                        'FØRK' L6,L7;

A7                  L6: PQUICKSØRT(A,J+1,U);
                        'GØTØ' L8;

A8                  L7: PQUICKSØRT(A,L,J-1);
                        'GØTØ' L8;

                    L8: 'JØIN' L6,L7;
                    'END'
A9                  'ELSE' PQUICKSØRT(A,J+1,U)
            'END'
        'END'
        'ELSE'
        'BEGIN'
            'CØMMENT' Linear Insertion Process. ;
FN          'FØR' I←L+1 'STEP' 1 'UNTIL' U 'DØ'
N-(FN+AN)   'IF' A[I] 'LT' A[I-1] 'THEN'
            'BEGIN'
DN              V←A[I];J←I;
EN+DN           'FØR J←J-1 'WHILE' A[J] 'GT' V 'DØ' A[J+1]←A[J];
DN              A[J+1]←V                                  EN
            'END'
        'END'
    'END';
```

PROGRAM 8

```
'PRØCEDURE' MATINV1(N,A,X,L,EPS);
'CØMMENT' Procedure evaluates the inverse of a real symmetric matrix A
    using the first order implicit iterative process (Chapter 7) with
    an initial approximation of I. On input, matrix X holds the initial
    approximation I and on exit, the inverse of A. The iterative
    process is terminated when the difference between successive
    approximations is less than 2*EPS* (the element of X with largest
    modulus). ;
'INTEGER' N,L;'REAL' EPS;'ARRAY'A,X;
'BEGIN'
    'REAL' C,D,XMAX,ZMAX,E;
    'INTEGER' I,J,K;
    'ARRAY' B[1:N,1:N],Y[1:N];
    L←Ø;

    'CØMMENT' Formation of X.A;

L1: 'FØR' I←1 'STEP' 1 'UNTIL' N 'DØ'
    'FØR' J←1 'STEP' 1 'UNTIL' N 'DØ'
    FO1ARA(1,1,N,O,O,X[I,K],A[K,J],K,B[I,J],D);
    XMAX←ZMAX←O;
    'FØR' J←1 'STEP' 1 'UNTIL' N 'DØ'
    'BEGIN'

        'CØMMENT' Solution for Y. ;

        'FØR' I←1 'STEP' 1 'UNTIL' N 'DØ'
        'BEGIN'
            FO1ARA(1,1,I-1,-X[I,J],O,B[I,K],Y[K],K,C,D);
            Y[I]←-C/B[I,I]
        'END';

        'CØMMENT' Solution for X. ;

        'FØR' I←N 'STEP' -1 'UNTIL' 1 'DØ'
        'BEGIN'
            FO1ARA(I+1,1,N,O,O,B[I,K],Y[K],K,C,D);
            Y[I]←Y[I]-C/B[I,I];
            C←ABS(Y[I]);
            'IF' C 'GT' XMAX  'THEN' XMAX←C;
            C←ABS(Y[I]-X[I,J]);
            'IF' C 'GT' ZMAX 'THEN' ZMAX←C;
            X[I,J]←Y[I]
        'END'
    'END';
    L←L+1;
    D←ZMAX/XMAX;
    'IF' D 'GT' 2*EPS 'THEN' 'GØTØ' L1
'END';
```

PROGRAM 9

```
'PRØCEDURE' MATINV2(N,A,X,L,EPS);
'CØMMENT' Procedure evaluates the inverse of a real symmetric matrix A
    using the second order implicit iterative process (Chapter 7), with
    an initial approximation of I. On input, matrix X holds the initial
    approximation I and on exit, the inverse of A. The iterative
    process is terminated when the difference between successive
    approximations is less than 2*EPS* (the element of X with largest
    modulus).  ;
'INTEGER' N,L;'REAL' EPS;'ARRAY' A,X;
'BEGIN'
    'REAL' C,D,XMAX,ZMAX,E;
    'INTEGER' I,J,K;
    'ARRAY' B,F[1:N,1:N],Y[1:N];
    L←0;

    'CØMMENT' Formation of X.A;

L1: 'FØR' I←1 'STEP' 1 'UNTIL' N 'DØ'
    'FØR' J←1 'STEP' 1 'UNTIL' N 'DØ'
    FO1ARA(1,1,N,0,0,X[I,K],A[K,J],K,B[I,J],D);

    'CØMMENT' Formation of INV(D-U).  ;

    'FØR' I←1 'STEP' 1 'UNTIL' N 'DØ'
    'BEGIN'
        F[I,I]←B[I,I];
        'FØR' J←I+1 'STEP' 1 'UNTIL' N 'DØ'
        'BEGIN'
            FO1ARA(I,1,J-1,0,0,F[I,K],B[K,J],K,C,D);
            F[I,J]←-C/B[J,J]
        'END'
    'END';

    'CØMMENT' Formation of D.INV(D-L).  ;

    'FØR' J←1 'STEP' 1 'UNTIL' N-1 'DØ'
    'FØR' I←J+1 'STEP' 1 'UNTIL' N 'DØ'
    'BEGIN'
        Y[J]←F[J,J];
        FO1ARA(J,1,I-1,0,0,B[I,K],Y[K],K,C,D);
        Y[I]←-C/B[I,I];
        F[I,J]←-C
    'END';

    'CØMMENT' Formation of INV(D-U).[D.INV(D-L)].  ;

    'FØR' I←1 'STEP' 1 'UNTIL' N 'DØ'
    'FØR' J←1 'STEP' 1 'UNTIL' N 'DØ'
    FO1ARA('IF' I>J 'THEN' I 'ELSE' J+1,1,N,'IF' I>J 'THEN' 0
           'ELSE' F[I,J],0,F[I,K],F[K,J],K,F[I,J],D);

    'CØMMENT' Formation of I-[X.A].[INV(D-U).D.INV(D-L)]. ;

    'FØR' J←1 'STEP' 1 'UNTIL' N 'DØ'
    'BEGIN'
```

```
        'FØR' I←1 'STEP' 1 'UNTIL' N 'DØ'
        FOlARA(1,1,N,'IF' I=J 'THEN' -1 'ELSE' 0,0,B[I,K],F[K,J],
              K,Y[I],D);
        'FØR' I←1 'STEP' 1 'UNTIL' N 'DØ' F[I,J]←-Y[I]
    'END';

    'CØMMENT' Formation of X+[I-X.A.INV(D-U).D.INV(D-L)].X.  ;

    'FØR' I←1 'STEP' 1 'UNTIL' N 'DØ'
    'BEGIN'
        'FØR' J←1 'STEP' 1 'UNTIL' N 'DØ'
        FOlARA(1,1,N,X[I,J],0,F[I,K],X[K,J],K,Y[J],D);
        'FØR' J←1 'STEP' 1 'UNTIL' N 'DØ' F[I,J]←Y[J]
    'END';
    XMAX←ZMAX←0;
    'FØR' J←1 'STEP' 1 'UNTIL' N 'DØ'
    'BEGIN'

       'CØMMENT' Solution for Y.  ;

       'FØR' I←1 'STEP' 1 'UNTIL' N 'DØ'
       'BEGIN'
           FOlARA(1,1,I-1,-F[I,J],0,B[I,J],Y[K],K,C,D);
           Y[I]←-C/B[I,I]
       'END';

       'CØMMENT' Solution for X.  ;

       'FØR' I←N 'STEP' -1 'UNTIL' 1 'DØ'
       'BEGIN'
           FOlARA(I+1,1,N,0,0,B[I,K],Y[K],K,C,D);
           Y[I]←Y[I]-C/B[I,I];
           C←ABS(Y[I]);
           'IF' C 'GT' XMAX 'THEN' XMAX←C;
           C←ABS(Y[I]-X[I,J]);
           'IF' C 'GT' ZMAX 'THEN' ZMAX←C;
           X[I,J]←Y[I]
       'END'
   'END';
   L←L+1;
   D←ZMAX/XMAX;
   'IF' D 'GT' 2*EPS 'THEN' 'GØTØ' L1
'END';
```