Parallel algorithms for asynchronous multiprocessors

This item was submitted to Loughborough University’s Institutional Repository by the/an author.

Additional Information:

- A Doctoral Thesis. Submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of Loughborough University.

Metadata Record: [https://dspace.lboro.ac.uk/2134/13616](https://dspace.lboro.ac.uk/2134/13616)

Publisher: © Nadia Yacob Yousif

Please cite the published version.
<table>
<thead>
<tr>
<th>AUTHOR/FILING TITLE</th>
<th>You, S. F., N.Y.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESSION/COPY NO.</td>
<td>003600/02</td>
</tr>
<tr>
<td>VOL. NO.</td>
<td></td>
</tr>
<tr>
<td>CLASS MARK</td>
<td></td>
</tr>
<tr>
<td>LOAN COPY</td>
<td></td>
</tr>
</tbody>
</table>

- 6 Jul 1986

30 Jun 1989

- 6 Jul 1990

- 5 Jul 1991

31 Jul 1991
PARALLEL ALGORITHMS FOR ASYNCHRONOUS MULTIPROCESSORS

BY

NADIA YACOB YOUSIF, B.Sc., M.Sc.

A Doctoral Thesis
Submitted in partial fulfilment of the requirements
for the award of Doctor of Philosophy
of the Loughborough University of Technology
October, 1983.

Supervisor: PROFESSOR D.J. EVANS, Ph.D., D.Sc.,
Department of Computer Studies

© Nadia Yacob Yousif, 1983.
DECLARATION

I declare that this 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 higher degree.

NADIA YACOB YOUSIF.
DEDICATED TO

MY PARENTS

"Do good, O LORD, to those who are good,
and to those who are upright in their hearts".

(Psalm 125, Verse 4).
ACKNOWLEDGEMENTS

The author wishes to express her deepest and sincere gratitude to Professor D.J. Evans for the friendly and considerate guidances, advice and willingness to assist at any time throughout the preparation of this thesis.

The author acknowledges also with gratitude the IRAQI government especially BASRAH University for their financial support to complete this project.

A sincere gratitude and thanks the author wishes to present for her parents, brothers and sisters for their patience and moral encouragement.

Acknowledgements are also presented to the following people:

- The parallel processing group in the Department of Computer Studies who helped to keep the parallel system reliable to complete the experimental part of this thesis.

- Miss Ana Filipe who offered valuable suggestions in the preparation of the graphics presented in this thesis.

- Miss J.M. Briers who typed this thesis with great efficiency.
I'm supposed to know something about science;
but I know nothing except the mathematics
it involves.
I can make calculations for engineers, electricians,
insurance companies, and so on,
but I know next to nothing about engineering or
electricity or insurance.
I don't even know arithmetics well.

GEORGE BERNARD SHAW.
The work presented in this thesis is mainly involved in the design and analysis of asynchronous parallel algorithms that can be run on MIMD type parallel computers, in particular the NEPTUNE system at Loughborough University.

Initially, different types of existing parallel computers including the Data-Flow computers and VLSI technology are described from both the hardware and implementation points of view. Basic ideas of programming such computers are also outlined. Also, the main characteristics of the NEPTUNE MIMD-system are presented together with the principles of synchronisation, the resource demands and the overhead costs of the parallel control structures. Such information is measured frequently in the performance analysis of the algorithms presented in this thesis in order to exploit the potentiality of the NEPTUNE system and parallel computers in general. The Speed-up and Efficiency factors are calculated and the optimum number of processors and processes is suggested in most of the algorithms presented.

This study includes both numerical and non-numerical algorithms. In the former topic, the iterative methods for solving linear systems of equations derived from the Dirichlet problem are investigated in parallel form where the Gauss-Seidel and S.O.R. algorithms are implemented with and without synchronisation. A parallel 4-point block iterative method is developed where the asynchronous SOR method is used. Here, the blocks are grouped in such a way that each process contains its own group, therefore, they can run in parallel. This method presents a high
level of parallelism. Finally, an analysis study of these algorithms is carried out by using Queueing Theory and Order Statistics.

In the non-numeric algorithms, the problems of sorting and merging are investigated. For example, the two-way merge and the odd-even merge methods are studied where the number of processes is greater or equal to the number of processors. The Jump Search algorithm is also developed as a merging method where different strategies of the Jump Search are implemented and compared. A comparison between these strategies and the Binary Search merging method is also presented. In addition, many sorting problems are investigated, in particular, the sorting by distributive partitioning method of Dobosiewicz [1978] which is compared with the parallel Quicksort. The analysis technique of all these methods requires a probability approach where some probability distribution functions are considered.

In this thesis, comparisons between a particular algorithm when implemented using different strategies as well as comparisons between different algorithms are also presented. These comparisons are obtained from a detailed analysis of the complexity of the algorithm.

The thesis concludes with a chapter summarizing the main results and suggestions for further study.
# CONTENTS

## ACKNOWLEDGEMENTS

<table>
<thead>
<tr>
<th>Chapter 1:</th>
<th>INTRODUCTION TO PARALLEL MACHINE ARCHITECTURES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>1.2</td>
<td>Pipeline (Vector) Machine</td>
</tr>
<tr>
<td>1.3</td>
<td>SIMD Machine</td>
</tr>
<tr>
<td>1.4</td>
<td>MIMD Machine</td>
</tr>
<tr>
<td>1.5</td>
<td>The NEPTUNE System</td>
</tr>
<tr>
<td>1.6</td>
<td>Data-Flow Systems</td>
</tr>
<tr>
<td>1.7</td>
<td>VLSI Systems</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 2:</th>
<th>PRINCIPLES OF PARALLEL PROGRAMMING AND LANGUAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2.2</td>
<td>Parallel Programming Concepts</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Explicit Parallelism</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Parallel Computational Models</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Implicit Parallelism</td>
</tr>
<tr>
<td>2.3</td>
<td>Programming the NEPTUNE System</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 3:</th>
<th>THE STRUCTURE OF PARALLEL ALGORITHMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>3.2</td>
<td>The Structure of Algorithms for SIMD and Pipeline Computers</td>
</tr>
<tr>
<td>3.3</td>
<td>The Structure of Algorithms for the MIMD Computers</td>
</tr>
<tr>
<td>3.4</td>
<td>The Structure of Algorithms for VLSI Systems</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 4:</th>
<th>THE DESIGN AND ANALYSIS OF ASYNCHRONOUS ALGORITHMS FOR LINEAR EQUATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>4.2</td>
<td>Basic Definitions</td>
</tr>
<tr>
<td>4.3</td>
<td>Methods for Solving Linear Systems of Equations</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Direct Methods</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Iterative Methods</td>
</tr>
<tr>
<td>4.4</td>
<td>Mathematical Notations for the Asynchronous Iterative Methods</td>
</tr>
<tr>
<td>4.5</td>
<td>The Convergence Theorem of Asynchronous Iterative Methods</td>
</tr>
<tr>
<td>4.6</td>
<td>Finite Difference Approximations to Derivatives</td>
</tr>
<tr>
<td>4.7</td>
<td>Theoretical Determination of the Optimum Relaxation Factor for the Iterative Methods</td>
</tr>
<tr>
<td>4.8</td>
<td>Rate of Convergence of the Basic Iterative Methods</td>
</tr>
<tr>
<td>4.9</td>
<td>Iterative Methods on the NEPTUNE Asynchronous System</td>
</tr>
<tr>
<td>4.9.1</td>
<td>Experimental Results with the Two-Dimensional Dirichlet Problem</td>
</tr>
<tr>
<td>4.9.2</td>
<td>Experimental Results with the One-Dimensional Dirichlet Problem</td>
</tr>
<tr>
<td>4.10</td>
<td>Parallel Iterative Methods with Over- and Under-relaxation Factors</td>
</tr>
<tr>
<td>4.11</td>
<td>The Behaviour of the Under- and Over-Relaxation Algorithms</td>
</tr>
<tr>
<td>4.12</td>
<td>The Analysis of the Asynchronous Iterative Algorithms</td>
</tr>
<tr>
<td>4.13</td>
<td>Conclusions</td>
</tr>
</tbody>
</table>

**Chapter 5: PARALLEL IMPLEMENTATIONS OF THE FOUR-POINT BLOCK EXPLICIT METHOD**

| 5.1    | Introduction                                                                   | 216 |
| 5.2    | The Basic Four-Point Block Iterative Method                                    | 218 |
| 5.3    | Experimental Results of the Parallel Four-Point Block Iterative Method         | 224 |
| 5.4    | Performance Analysis of the 4-Point Block Methods and the Standard Point Method | 247 |
| 5.5    | Conclusions                                                                    | 260 |

**Chapter 6: COMPARATIVE STUDIES OF PARALLEL SORTING AND MERGING ALGORITHMS**

| 6.1    | Introduction                                                                   | 262 |
6.2 Sorting Algorithms
   6.2.1 The Bubble Sort
   6.2.2 The Neighbour Sort
6.3 Merging Algorithms
   6.3.1 The Parallel 2-Way Merge Algorithm
   6.3.2 The Parallel Odd-Even Merge Algorithm
   6.3.3 The Parallel New Merge Algorithm
   6.3.4 Merging by the Binary Search Algorithm
   6.3.5 Merging by a Parallel Jump Searching Algorithm
6.4 Conclusions

Chapter 7: SORTING METHODS BY PARALLEL DISTRIBUTIVE PARTITIONING ALGORITHM
   7.1 Introduction
   7.2 Parallel Quicksort Algorithm
      7.2.1 The Quicksort Analysis
   7.3 Parallel Distributive Partitioning Sorting Methods
      7.3.1 The Parallel DPS: Version 1
      7.3.2 The Parallel DPS with Merging Algorithm: Version 2
      7.3.3 The Performance and Overhead Measurements of the Algorithms
   7.4 Conclusions

Chapter 8: CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

APPENDIX A

REFERENCES
CHAPTER 1

INTRODUCTION TO PARALLEL MACHINE ARCHITECTURES
1.1 INTRODUCTION

With the advance of technology, the computation speed of electronic computers becomes a matter of interest for both the designer and the user of the computers who are generally interested in solving problems quickly and economically. As a matter of fact the computation's cost should be decreased as technology advances. This decrease can be achieved by increasing the computations speed which in turn can be improved by making the electronic components of the computer work faster or to make the current circuitry much denser.

In actual fact, since the earlier times of computers where the electronic components were very slow such as relays and vacuum tubes which were introduced in the 1940's and early 1950's the demand for faster ones was appreciated. Therefore, in the late 1960's Integrated Circuits (IC) were used followed by the Large-Scale Integrated (LSI) circuits which consist of thousands of transistors and related components. Recently the Very Large Scale Integrated (VLSI) circuits, to be discussed in Section 1.7, are designed where a large number of problems can be solved with.

Unfortunately, it seems difficult or costly to increase the speed by improving the electronic components at the same pace any further. However, due to basic physical laws it seems that we are approaching the upper limit of the speed at which a digital computer can transfer information. In order to explore this consider the example of adding two 32 bit-words. This requires about 300 milliseconds on a relay computer in 1944, 300 microseconds on a tube computer in 1954 and 300 nanoseconds on CDC 6600 computers in 1964. Suppose, now, we try to establish a computer with an addition time of 300 picoseconds. Then, as the speed at which an electrical signal travels is 0.03 cm/psec, it would propagate only 9 cm during the entire execution time of the add instruction. Thus, it will be difficult to achieve such small
propagation delays or reduce the distance so much.

An attempt to increase the speed must be made to find another solution rather than always reducing the circuit execution or propagation time.

One such approach is to introduce parallelism where many identical processing elements are connected together hoping that connecting \( P \) processors will potentially increase the overall speed approximately \( P \) times that of a single computer. Although this upper bound can rarely be achieved in practice due to some reasons to be discussed later in the thesis.

Flynn [1966] had classified the parallel computers into four types. The Single Instruction Stream Single Data Stream (SISD) computer which is the sequential computer, the Single Instruction Stream Multiple Data Stream (SIMD) computer, the Multiple Instruction Stream Single Data stream (MISD) which can be considered as a pipeline computer, and the Multiple Instruction Stream Multiple Data Stream (MIMD) computer.

Another classification approach is introduced by J. Murtha and R. Beadles [1964] where the parallel processors systems are classified into three types:

1. general purpose network computers,
2. special purpose network computers characterized by global parallelism,
3. non-global computers where each module is only semi-independent (i.e. local parallel).

General purpose network computers were further divided into:

1. parallel networks with a common central control,
2. parallel networks with many identical processor elements which can independently execute instructions.

Special purpose network computers were divided into:
1. pattern processors
2. associative processors

In the following sections we shall discuss the pipeline, the SIMD and the MIMD computers by outlining their main architecture differences and the advantages of each model.
1.2 PIPELINE (VECTOR) MACHINE

The pipeline computer architecture has received considerable attention since the 1960's when faster and more cost-effective systems became critical.

Pipelining is one form of embedding parallelism or concurrency in a computer system. Its basic idea is a segmentation of a computational process into several sub-processes which are executed by dedicated autonomous units (pipelining segments) which operate concurrently.

Some machines such as the Texas Instrument TI ASC (Watson, 1972), CRAY-1 (Cray 75) and the CDC STAR-100 (Hintz R.G. and Tate, D.P., 1972) have distinct pipeline processing capabilities, either in the form of internally pipelined instruction and arithmetic units or in the form of pipelined special purpose functional units. Ramamoorthy and Li (1977) represented many theoretical considerations of pipelining and represented a survey of comparisons between various pipeline machines that operate in either sequential or vector pipeline mode whose trade-off was studied. Also a top-down, level-by-level characterization of pipeline applications in computers and the associated configuration control were explained in this reference.

The simplified model of the pipeline machine is shown in Figure 1.1. This model consists of \( k \) pipelining segments. Each segment executes a part of the processing and the result appears at the end of the last segment. Since buffers separate the segments and the transfer between segments can be effected only after all the buffers have stabilized, therefore, the expected full speed-up is not always achieved.

The main characteristic of the simplest pipelining is that it may overlap the Instruction (I) and the Execution (E) cycles of computers. For example, consider the process of executing an instruction. Normally, it involves fetching the instruction, decoding the operations, fetching the operands before it is finally executed. If this process is decomposed into
FIGURE 1.1: Pipeline processor system
these four subprocesses and executed on the four modules shown in Figure 1.2(a), then four successive independent instructions may be executed in parallel. Specifically, while the EXEC module is executing the first instruction, the Operand Fetch (OF) module fetches the operand needed for the second instruction, the Instruction Decode (ID) module prepares the operations for the third instruction, and the Instruction Fetch (IF) module fetches the fourth instruction. The overlapped execution among the four modules is illustrated as a space-time diagram in Figure 1.2(b). From this diagram one can observe how independent instructions are executed in parallel in a pipelined machine.

![Figure 1.2(a): Pipelined processor to execute an instruction](image)

![Figure 1.2(b): Space-time diagram](image)

Meanwhile with dependent instructions, their input and traversal through the pipe have to be paused until the dependency is resolved. Thus, an efficient internal bussing structure is needed to route the results to the required segment efficiently.

On the other hand, branching is quite damaging to the pipeline performance than instruction dependency. Thus, when a conditional branch
is encountered, one cannot tell which sequence of instructions will follow until the deciding result is available at the output. Therefore, a conditional branch not only delays further execution but also affects the entire pipe starting from the instruction fetch segment.

To remedy the effect of branching, different techniques were employed to provide mechanisms so that processing can resume even if an unexpected branch occurs (Ramamoorthy and Li, 1977).

Also, the pipeline approach can speed-up arithmetics and the pipelining advantages will be obtained when floating point operations are used since they are long. For example, the IT ASC computer has an eight-segment pipeline, as shown in Figure 1.3, for executing floating point instructions.

The pipeline is full when all segments are fully used, whereas in the floating-point addition some of the segments are not used (i.e. segment 6,7). This is one case when the speed-up is reduced.

For vector processing in a pipeline machine with built-in vector instructions (e.g. CRAY-1), conflicts may arise in accessing arrays from the m parallel memories of Figure 1.1. If the data must be rearranged between the memory and processor, an alignment network is shown for this purpose in Figure 1.1.

Otherwise, the memory hierarchy may be a standard one.

Finally, to illustrate how best one can use pipeline computers we must examine the timings of the pipeline operations. Suppose that the subtasks
of an operation are designed so that each subtask is completed in time \( \tau \), the total time to complete an operation is \( \sigma \), and let \( t \) be the time to complete one instruction in a standard computer. Then, the time required to perform \( n \) pipeline operations will be \((n-1)\tau+\sigma\), since the first operation leads to a result in \( \sigma \) time, and the \((n-1)\) subsequent results appear at the output in \((n-1)\tau\) time. Therefore, to achieve a speed-up when performing \( n \) operations we require

\[
(n-1)\tau + \sigma < nt
\]

or

\[
n > \frac{(\sigma-\tau)}{(t-\tau)}
\]

From this inequality one should notice that the pipeline can be utilised if the vector of operations is long relative to the number of segments in the pipeline machine, and the above formula should be satisfied.
1.3 SIMD MACHINE

The SIMD machine or the array processor is a parallel computer which consists of \( P \) identical processors arranged in an array form. These processors are controlled by a single control unit, which would decode and broadcast the instructions to all processors within the array. Besides sharing the control unit, the processors share a memory hierarchy. This means that each processor has its own private memory which provides it with its own data stream. Hence each processor would perform the same instruction on its own data at exactly the same time, consequently this machine is known as a *synchronous* machine.

A SIMD computer of \( p \) processors is represented diagramatically in Figure 1.4. An alignment network that allows data to be transferred from one processor to another or back and forth between memories and processors is illustrated in this diagram.

![Diagram of SIMD processor machine](image)

**FIGURE 1.4:** SIMD processor machine

Several SIMD machines have been built and successfully used in
particular application areas. The ILLIAC IV (Barnes et al, 1968 and Kuck, 1968), designed at the University of Illinois and at Burroughs corporation and built by Burroughs, is a scientific-numerical machine with 64 fast, floating point processors of 64-bit words each.

The advantages of SIMD computers seem to be greatest when applied to the solution of problems in matrix algebra or finite difference methods for the solution of the partial differential equations. It is known that most algorithms in this area require the same types of operations to be repeated on large numbers of data items. Hence, the problem can be distributed to the processors that can run simultaneously. Typical finite difference meshes are (50×50) or (100×100) so such organisation would imply a requirement for 2500 or 10000 processor systems. When a three-dimensional (100×100×100) problem is to be tackled, $10^6$ or more processors could be needed. If the system has fewer processors than the mesh size it is necessary to make repeated operations in a fashion analogous to those completed on a serial computer. However, the large degree of parallelism in real problems would suggest to build a system of many more processors than the 64 of the ILLIAC IV system.

To build a system of a large number of processors leads to some obvious constraints on the type of processors:

1. they must be physically small,
2. they must have low power consumption and
3. they must be relatively cheap to make.

Examples of systems with the above properties are, the Goodyear Aerospace STARAN IV (Enslow, 1974) which is an associative processor of 1-bit words and is built in multiples of 256 processors. STARAN is considered as an associative array processor, because the data can be accessed in either the word direction or the bit slice-direction (Batcher, 1977).
The ICL Distributed Array Processor, DAP, (Reddaway, 1973) which is installed at Queen Mary College is also a SIMD computer made out of many low speed processors. However, the architecture of the DAP comprises of cellular array processors which enables very high computing power to be obtained for calculations which display a large degree of parallelism. DAP is composed of 4096 processors arranged in a (64×64) matrix and each processor receives a single instruction stream broadcast from a single unit called the master control unit.

The speed of bit-organised processors is relatively slow compared with modern high speed architectures. This is due primarily to the store access times. As it is fundamental to their design that the average memory processor transfer is only single bit then the operation times are ultimately controlled by the cycle times for access to the memory chips (Parkinson, 1976 and Parkinson, 1982). Therefore, the actual speed-up achieved by a SIMD machine of such processors is significantly reduced.

As in the pipeline machine (Section 1.2), the conditional statement creates problems for SIMD machines. This allows the hardware designer to propose a design to remedy such a problem. Therefore, a very important facility is proposed to be added to the processors which is used as a switch to permit processors to ignore certain instructions transmitted by the main control unit or to inhibit the operation of some processors.

In addition to the main outline of the SIMD computers an important task is to discuss the interconnection problem. Obviously, a complete interconnection network, where each processor is connected to all other processors, is expensive and unmanageable by both the designer and the user of the system. Therefore some other interconnection patterns are proposed to be of primary importance in determining the power of parallel computers.

The interconnection pattern used in DAP or ILLIAC IV is that the
processors are arranged in a two-dimensional array where each processor is connected to its nearest 4 neighbours as illustrated in Figure 1.5.

![Figure 1.5: (p×p) array processor](image)

This type of connection is very suitable to solve two-dimensional partial differential equations.

Also processors can be arranged as a regular mesh (Thompson and Kung, 1977), in a perfect shuffle pattern (Stone, 1971) or in various special-purpose configurations for merging, sorting, or other specific applications. However, Goldschlager [1982] proposed a universal interconnection pattern which can simulate all these patterns in linear time.

In practice, when implementing a particular algorithm on one of the above-mentioned patterns, one should be very careful on matching the algorithm requirements and the pattern concerned in order to prevent any undesirable communication delay that may seriously affect the execution time of the algorithm and hence reduce the speed-up factor.

As it is necessary in SIMD computers to use a suitable interconnection pattern for a particular algorithm, and to use as many processors as possible so that a high degree of parallelism can be exploited therefore, this type of computer is called a special-purpose computer whose development is a matter of importance for the solution of those problems.
1.4 MIMD MACHINE

The MIMD machine is composed of a number of processors each with its own control unit. Thus, each processor is capable of generating its own instruction stream which can then be executed on its own data simultaneously. Hence, the MIMD machine is an asynchronous machine.

This is why the MIMD computer is considered as a collection of mini-computers or microprocessors and collectively as a multiprocessor system.

The basic diagram of a MIMD machine of p processors is illustrated in Figure 1.6. Although each processor includes its own control unit, a higher level control unit may be used to control the transfer of data and to assign tasks and sequences of operations between the different processors. Frequently, one of the processors may itself serve this higher level control function as in the NEPTUNE system (see Section 1.5).

Within the general class of MIMD systems there is a wide range of physical interconnection between the processors. Usually the processors, share the I/O equipment and the memory hierarchy. However, in Figure 1.6, an interconnection network is shown. This network allows various units to communicate data and each processor can transmit an interrupt signal to any other processor. Moreover there frequently occurs in such systems memory conflict which is one factor that reduces the performance of a multiprocessor system. The memory access conflict is of two types, software and hardware. A software memory access conflict occurs when a processor attempts to use data that is currently being accessed by another processor which has activated the "lock" to prevent any other processor from accessing the same data set. This data set is called a "critical section"; (see Chapter 2). Software memory conflicts are often known as "memory lockout". However, when a processor encounters a memory lockout, it waits and repeatedly checks the status of the lock until the "unlock" state is reached.
A hardware memory conflict occurs when two or more processors attempt to access the same memory module (or unit) simultaneously, i.e. the conflicting requests are made during a single memory cycle. Therefore, only one access can be made per memory cycle and the other requests must wait usually for a cycle or two in each case. The system will have a greater degradation when a large number of these conflicts occur. Such degradation is known as "interprocessor interference".

To reduce the memory conflicts and the processor interconnection, a "private memory" is then suggested to be associated with each processor in which important data is stored.
Usually in parallel systems the units are "tightly" or "loosely" coupled. However, the amount of synchronization and memory sharing between the processors is often an indication of how tightly coupled they are. Therefore, the SIMD machines are considered as tightly coupled machines and most of the MIMD machines are loosely coupled.

The most integrated, or tightly coupled systems comprise several processing units sharing the same memory while the most loosely coupled systems comprise totally separate computer systems linked by slow speed communication lines. Generally, the shared memory could be regarded as very fast and composed of several parallel memory units. Therefore, conflicts in accessing a particular unit would be very possible. However, the communication line providing the use of such memory is restricted to communication. Thus, the shared memory requires a different type of management from the communication line.

If we consider a tightly coupled system then we notice that there are two major limitations of the performance of the system. The first is the degradation due to conflicts to access the main memory or the I/O devices as discussed above. The second limitation arises in synchronizing and scheduling jobs on the multiple processors. However, one can expect a processor multiprocessor to exhibit a throughput which is substantially less than p and this is due to interference.

One approach to reduce conflicts and store clashing is that to add a private memory to each processor so that the processor can obtain data from its own memory and the common memory. On the other hand, because of interconnection and conflicts problems, it is envisaged that large numbers of processors cannot be utilised effectively in a multiprocessor system. Therefore, a small number of processors is always preferable in designing a MIMD system.

In fact, most real multiprocessor systems have at most 16 processors.
(e.g. the NEPTUNE system to be described in Section 1.5 has 4 processors). However, a system with two processors is much more common. The Interdata Dual system (Barlow et al, 1977) is one example of this type of computer system.

Most major manufacturers provide multiprocessors but none of them promise more than four processors in the mid-1970's, but attempts are proposed to increase the number of processors. Therefore, up to ten processors are provided in the very specialised Bell Laboratories, CLC, a radar signal processing control computer (Bell, 1975) and up to 16 PDP-11 processors are included in the experimental C.mmp at Carnegie-Mellon University (W.A. Wulf and C.G. Bell, 1972). This leads to the definition of the "highly parallel machines" which are machines with more than ten processors and the processors are integrated to allow them to work on the same algorithm. Therefore, a highly parallel machine is approximately defined as a multiprocessor with a large number of processors and hence it is a subclass of the multiprocessor systems (Comfort, 1963).

The interconnection network for a multiprocessor system with small number of processors is quite manageable. Otherwise, a simpler interconnection pattern must be used as we mentioned for the SIMD machine. However, one reduced network is the STAR configuration illustrated in Figure 1.7, where only one processor is connected to each of the remaining processors.

FIGURE 1.7: STAR configuration network
To illustrate some MIMD machine characteristics, we discuss now briefly the Interdata Dual processor system and the C.mmp and Cm* systems.

The Interdata Dual processor has been developed at Loughborough University. The model of this system consists mainly of two identical processors which are Interdata model 70. Each processor has 32kb of private memory and shares a 32kb shared memory as in Figure 1.8.

![Diagram of Interdata Dual System](image)

**FIGURE 1.8:** Interdata Dual System

The first 32kb of processor B is not accessible by processor A while the second 32kb is accessible by both processors.

This system has the property of asymmetry, that is when processor B is accessing the common or its private memory, processor A remains locked out of the common memory until that accession is completed. Whereas, if processor A is accessing the common memory then processor B is locked out of both its private and common memory until the memory cycle of A is completed.

Actually, both processors are subject to a delay of one memory cycle time which is 1 microsecond due to memory contention. However, due to the memory bus interface logic, processor A reserves the common memory 0.5 microsecond before it requires it and so processor B is, in fact, subject to a maximum delay of 1.5 microseconds due to memory contention. On the
other hand, processor A suffers more than processor B because it has a minimum delay of 1 microsecond while processor B has zero microseconds as the minimum delay. The performance measurements of the Interdata dual system has been presented by Barlow and Evans, [1977].

The second system, the C.mmp system, represented in Figure 1.9 has been developed at Carnegie-Mellon University. There are two cross-bar switches, switch 1 allows each processor to communicate with all primary memories, and switch 2 connects each processor \( P_i \) with the various controllers. These controllers manage all the secondary memories and I/O devices.

Each processor is actually a model of a DEC PDP-11 and it is a complete computer with its own local memory and controllers for secondary memories and devices. In the figure, Dmap refers to the Data operations component that is associated with each processor and is used to translate addresses at the processor into physical memory addresses for all accesses to the shared memory, since the address space of the main memory greatly exceeds that of the PDP-11 itself.

![Diagram of the C.mmp system]

**FIGURE 1.9:** The C.mmp system
The shared primary memory consists of up to 16 modules $M_0, \ldots, M_{15}$. Each module consists of 64k, 16 bit words. However, the size of the physical shared memory is $2^{20}$ words (or $2^{21}$ bytes).

The problem of memory interference in this multiprocessor system has a high cost effectiveness if 15-30 processors are used. However, conflicts from accessing the memory are reduced to a minimum by placing a "cache memory" between the Dmap switch 1. This will help to diminish the number of requests for a single memory and will provide faster access.

In this system, the local memories are used only for interrupt and error-recovery, but not for programs or data, the common memory is only available for programs so that memory interference may be quite severe. However, to obtain reliability in C.mmp, the interleaving is done on the high-order bits, so that all the instructions of any program are in the same block with an additional degradation. On the other hand, the routing of memory requests from the processors to the shared memory is done by the hardware within the cross-bar switch. This, however, increases the price of the switch even more, lowers the reliability because of complicated hardware and reduces the performance speed because of queueing delays in the switch. Finally, because of the cross-bar switch and the limited use of the local memories, the inherently sequential problems cannot be implemented efficiently on such systems.

Another system that has been developed at Carnegie-Mellon University is the Cm* system (Swan, et al, 1975). This system is a multiprocessor system which is composed of "computer modules", each consisting of a DEC LSI-11, a standard LSI-11 bus, a local memory and devices. All primary memory in the system is potentially accessible to each processor. Within each module there is a local switch called "Slocal" which is used to route the memory references from and to the local memory of the processor to and
from the other modules through the "Map bus" or through the Kmap controller to other processor clusters. The term "cluster" is used to comprise a group of computer modules. The hardware would accept a maximum number of processors to be 14 in each cluster; the general issue is how many processors the controller-bus combination can support before approaching saturation. The clusters are connected via intercluster buses. A Cm* configuration can have an arbitrary number of clusters as shown in Figure 1.10.

FIGURE 1.10: The current Cm* configuration

The advantage of using a controller is that to ensure mutual exclusion on accessing shared data with minimum overhead. However, this map and its map-bus are the critical shared resources that may lead to deadlocks. On
the other hand, this system has some disadvantages. First, since each module is connected to an I/O device, then if a module is lost, all I/O devices attached to it are lost. Therefore, this is not very desirable if we want the system to be desirable. The second disadvantage is that the data of the local memories are shared by other modules. This will create the locking problem and will cause delays because, the transfer is performed by packet switching. These delays are not preferable in a process-control system. Finally, this system is not very suitable for off-line problems that cannot be partitioned into independent subproblems.

An alternative proposal for an MIMD computer is due to Flynn, et al [1970]. He proposed to interconnect several independent processors, each of which executes an independent instruction stream. The proposal is to make processors "skeleton" processors, by removing from them all arithmetic functions and computational logic where the functions are performed by highly specialised high-speed processors that are shared amongst the skeleton processors as shown in Figure 1.11. This sharing is done by closely synchronised time-phased switching, thus the resultant system avoids many of the contention problems associated with shared resource systems.

When a skeleton processor generates an instruction, it will obtain the operands for this instruction and requests access to a high-speed arithmetic unit that matches the instruction. If one arithmetic unit is available, the operation is performed and the result returned to the skeleton processor. In the case of conflicts, the request for computation can be placed in a queue or the request can be repeated until an arithmetic unit is free.

The advantage of this system is that it creates a multiple-instruction stream processor, but the expensive components of the processors are not replicated to the same extent that the processors are replicated.
FIGURE 1.11: MIMD machine with skeleton processors and centralized computation facilities

Also, the utilization of this system is high because of the sharing between processors and a high speed computation is obtained from the specialized arithmetic units.

Finally, the major advantage of the MIMD systems with several identical processors is the greater reliability. This means, the system continues to operate even when one memory unit or one processor fails. However, reliability is not compatible with increasing throughput and greater execution speed. Newman [1982] has presented some considerations for the reliability of parallel systems and of the systems at Loughborough University. His paper also presented some theoretical investigations to detect and recover from failure of one or more processors in a system with several processors and some shared memory.

Finally, to obtain a highly reliable parallel system from the hardware point of view, it may be possible to characterise the hardware organisation
by the nature of the system utilized to interconnect the processors, memories and the I/O devices. Three basic interconnections for the multi-processor systems which are:

(1) time shared buses;
(2) crossbar switch matrix; and
(3) multibus, multiport memories,

have been discussed by Enslow [1977] and [1974]. The C.mmp and C*m* described earlier are organised according to the above second strategy. Kuck [1978] has described many computer organisations concentrating on various parts of the computer, which are: processors, control units, main memory, interconnection networks and memory hierarchies. Moreover, many computer models are included in the design of parallel computers which have different structural features.

The following section will describe the current system at Loughborough University which is configured from the Texas Instruments 990/10 models.
1.5 THE NEPTUNE SYSTEM

Another type of MIMD machine to be described in this section is the NEPTUNE system which has been developed at the Department of Computer Studies of Loughborough University (Barlow, et al (1981)). This system is the vehicle used to implement all the parallel algorithms represented in this thesis.

The NEPTUNE system comprises four Texas Instruments 990/10 minicomputers configured as shown in Figure 1.12. This type of minicomputer offers a memory mapping feature with memory capacities up to one million 16 bit words. The instruction sets include both 16-bit and 8-bit byte addressing capability. No floating point arithmetic is available in the hardware.

The system consists of four processors each with its private memory of 128kb and one processor (P0) has a separate 10 Mb disc drive. A local bus (or TILINE), by which the access to the local memory is made, is attached to each processor. Each of the four local TILINEs is attached, via a coupler, to a fifth shared TILINE which is attached to a shared memory of 64kb and a 50Mb disc drive, therefore each processor can access 192kb of memory. This is because, the TILINE coupler is arranged so that the shared memory follows continuously from the local memory of each processor. Also, the 50Mb shared disc can be accessed by each processor with disc interrupts being transmitted to each.

Each processor runs under the powerful DX10 uniprocessor operating system which is a general-purpose, multitasking system. It features an effective file management package, which includes support for multi-key indexed files. It is also a multi-terminal system capable of allowing each user the ability to have exclusive control of the system.
The DX10 operating system requires: (1) the 990/10 or 990/12 CPU with the mapping option and a minimum random access memory of 64 16-bit words, (2) a system disc, (3) a Video Display Terminal (VDT), and (4) a means for disc backup. The disc backup may be magnetic tape or a second disc drive. Several disc models and disc management concepts can be found in [Texas Instruments, 1].

![Diagram of the current NEPTUNE configuration](image)

**FIGURE 1.12:** The current NEPTUNE configuration

Meanwhile, the DX10 operating system has been modified at Loughborough University to permit the processors to run independently and yet to cooperate in running programs with data in the shared memory. The co-
operation is suggested because, the shared 64kb memory belongs to processor P₀, i.e. only P₀ is permitted to allocate blocks within the shared memory, and a small area of memory at the head of the shared memory is reserved for the inter-processor communication where this area is used to inform processors P₁, P₂ and P₃ of any shared memory allocations affecting them.

The logical composition of a parallel program to be implemented on this system consists of two parts. One part contains the program code and the local variables; the other part contains the shared variables. As the 990/10 hardware allows a program to exist in three segments. Then, the commands used to generate the parallel programs ensure that these two parts of the program reside in different segments. Therefore, when a parallel program is executed by processor P₀, the segment containing the shared variables is loaded into the shared memory and the communication area is set to contain pointers to that segment. When the other processors execute the same program (or task) the non-shared segments of the task are then loaded into the private memory of each processor and the shared segment addresses are taken from the communication area. It follows that processor P₀ must always execute the parallel task.

Now, from the user's point of view, the user's interface to the NEPTUNE system is provided by the System Command Interpreter (SCI). This provides several ways in which commands may be issued. However, when a command is typed, the SCI will respond for each command parameter required and performs some checking on the value given.

DX10 supports a tree-structured filing system. Each installed disc pack which is known as "volume" is maintained independently. However, the filename must be specified with the name of the volume on which this file exists.
One of the facilities of the SCI is the synonyms that are not supported within DX10. The synonym is used to ease the typing of file-names and it is usually a short character string used to represent another long string. Many commands concerning the file are available in the SCI such that to add to, delete from and list the contents of the file from the specified directory containing such a file.

In the Texas Instrument 990/10, the foreground and the background features are available for a task to be run. If a user executes a task in the foreground, then no other command can be enhanced at that time until the foreground execution is finished. On the other hand, several tasks and commands may be accepted from the terminal in the background situation. However, background tasks should not involve I/O with the terminal. Commands are available for inspecting the state of the background.

For file and directory operations we refer to the reference manual [Texas Instrument, 2 & 4]. However, to edit a new file, the XE command must be used and the QE command is used to Quit Editing. In this stage, the user should give a name to the file just edited. When some modifications are made to an existing file, the user should replace the new version of the file onto the old one and this is done by the SCI when the user responds 'YES' to the SCI.

The four processors in the NEPTUNE system have a slightly different memory access time to the shared memory. However, the access to the local memory of each processor is ~0.6 µseconds. The accesses to the shared memory made by processors P₀, P₁, P₂, P₃ are: 0.81 µsecs, 0.52 µsecs, 0.71 µsecs and 0.72 µsecs, respectively. The total access time is the sum of the access to the local and the shared memory made by each processor. Although the processors are identical in many hardware features,
they are different in their speed also. The relative speed of processors \( P_0, P_1, P_2 \) and \( P_3 \) are 1.0, 1.037, 1.006 and 0.978 respectively. This will, however, reduce the efficiency of the system and decrease the performance measurements of an algorithm with synchronization.

The research on this system is still under further consideration and many software aspects are being improved. For example, attempts are made to allocate any processor to a particular terminal when a user logs in to that terminal. Also, instead of having \( P_0 \) to be the controller of the system (i.e. the main I/O activities and the compilation of the program are all performed on processor \( P_0 \)), the system is now capable of generating the SCI command of compilation on each processor.

The above-mentioned discussion includes only the hardware features and the operating system of the NEPTUNE MIMD system. The related programming concepts are left to be discussed in Chapter 2.
1.6 DATA-FLOW SYSTEMS

In all the parallel processing systems discussed in the previous sections, many computers are to be used in the organisation of such systems, therefore it is necessary to have a system-wide architecture and a computational model which they will all obey. Meanwhile, the general-purpose organisations for interconnecting and programming these systems which are based on the sequential control flow (Von Neumann) model have not been very forthcoming. In other words, these parallel systems are said to have a control mechanism which defines how one computation affects the execution of another computation. This means that the control is signalled to one computation if an input is required or if the output from one computation is required to the other computation.

For another approach, a question has been raised and that is which parallel organisation will be contributed to the design of a new general-purpose organisation. The answer is to manipulate a data mechanism which defines the way a particular argument is used by a group of computations, i.e. a separate copy is given for the value of each argument shared amongst many computations and an argument can also be used by having a reference to it stored in each accessing computation. This leads to the notion of Data-Flow which is a programming representation that follows the intensive studies of both the concurrent operation within a given computer system and the representation of parallelism in a programming language. The execution of a data-flow program is known as data-driven.

Although the research in the data-flow area has concentrated on the programming language aspects rather than on the implementation of data flow machines, this does not mean that no attempt has been made to build a data-flow machine. In fact, J. Dennis started data flow research in 1968. A few years later, a processor had been developed that could achieve a highly
parallel execution of data-flow programs (Dennis J.B. and Misunas D., (1975)). Currently, data flow computers are operational at the University of Utah, and Manchester and projects at MIT and University of California (UC), Irvine have shown, via simulation, that such an approach is viable. Actually, a primary motivation for studying data-flow is the advent of LSI technology which makes feasible the construction of a general purpose computer comprised of hundreds of asynchronous operating processors. In this system a data-flow program can be partitioned into small tasks that may be executed asynchronously by independent processors and in a data-driven manner.

In addition to the asynchronous, data-driven style of execution, data-flow is conceptually memoryless. All values are carried by tokens exchanged between operators. Thus all calculations are on values rather than on the contents of the addressable memory cells. This implies a purely functional and side-effect-free execution. However, the information must be supplied to a program explicitly in the form of arguments where the arguments propagate through the program graph causing the final resulting values to be returned to the caller of the program. This leads to a protection principle proposed by such a system. This principle is that no program can ever gain access to any information that is not explicitly passed to it by the caller as an argument. A protection mechanism and its implementation in a data-flow system has been developed by Bic [1982].

Now, we describe the principles of a data-flow system architecture. The processor proposed by Dennis and Misunas [1975] has a basic architecture which avoids the problems of processor switching and the processor-memory interconnection. Also it has the advantages of using the pipelining, a simple representation and the implementation of concurrent activity. The
structure of such a processor is shown in Figure 1.13.

The memory, in which the data-flow program is stored, consists of many instruction cells, each of them corresponding to an operator of the data-flow program. Each Instruction Cell is composed of three registers. The first register holds the instruction which specifies the operation to be performed. The second and third registers hold the operands for use in the execution of the instruction.

Following Figure 1.13, a cell is enabled and signals the 'Arbitration Network' if it has an instruction and the appropriate operands. The Arbitration Network transmits its contents as an operation packet to an 'Operation Unit' which performs the required function. The result of an operation is formed as one or more data packets. It consists of the computed value and the address of a register in the memory to which the
value is to be sent. The 'Distribution Network', however, accepts these data packets from the Operation Units and utilizes the address of each to direct the data item through the network to the correct register in the memory.

In this structure, many Instruction Cells may be enabled simultaneously, and it is the main job of the Arbitration Network to efficiently send the operation packets to the Operation Units and to queue the operation packets that are waiting for each Operation Unit. Since the Arbitration Network has many input ports and only a few output ports, the rate of the operation packets flow will be greater at the output ports. Therefore, to minimize the number of connections to the memory, a serial representation of packets is necessary at the input ports, but a more parallel representation is required at the output ports so a high throughput may be achieved. Also, the Operation Units are organised as a pipeline in order to increase the throughput.

Eventually, the structure of the above data-flow processor is the same as that of a MIMD system. It is hierarchically constructed as a network of simple modules interconnected asynchronously. Each module has a local memory and a pipeline of logical units to execute several concurrently active instructions [Rumbaugh, J., 1977]. This design has a lot of promise in a new technology. To date it has not achieved its full potential. Perhaps when the design has been fully explored it may produce the expected promise.
1.7 VLSI SYSTEMS

We discussed in Section 1.1 how computer design is currently developing which has led to the introduction of the ICs. However, by the mid 1970s, Large Scale Integrated (LSI) circuits that contain several thousand transistors and related components were in use. This has led to the existence of microprocessors where a whole computer can be assembled on a semiconductor chip. The actual number of components in one chip depends on the speed of the devices and the regularity of the pattern used to lay them out on the chip.

Following the rapid advances in LSI technology, the Very Large Scale Integrated (VLSI) circuits were introduced where the number of transistors that the LSI circuits contain will be increased by another factor of 10 to 100 in the next 10 to 20 years (Mead and Conway, 1980). Particularly, by the late 1980's it became possible to fabricate chips containing millions of transistors. The devices and interconnections in such VLSI systems have linear dimensions smaller than the wavelength of the visible light.

Generally, chip design requires computer-aided design assistance because of the enormous number of computers, packaging, and manufacturing problems which are so far unsolved. This leads to the prediction that VLSI will only be used in memories which are easier to design since the patterns are extremely regular. This is partially confirmed by the LSI experience which shows that the number of components on memory chips can be as much as five times that of a processor chip.

The number of bits per chip has been quadrupling every few years; in the mid 1970's, technology passed through the era of 1k-, 4k- and 16k-bit memory chip. In 1981, the number has expanded to 32k-bit and it is predicted that the chip will have 64k-bits capacity by 1985.

The advanced technology involved predicts radical changes in the chip
design and computer system architectures. Parallel processing on a single chip, in either a MIMD, SIMD or pipelining fashion, is advocated. The design of these chips will rely heavily on the regularity of patterns for communication between active elements. Meanwhile, the study of the VLSI circuits is not limited to the hardware aspects but also software tools become more and more important in the design and testing stages. This means that the production of a new chip requires software as well as hardware engineering knowledge. However, computer scientists have developed and are still developing more new algorithms and new communication techniques to exploit the potentiality of the VLSI system.

More recently, a considerable research interest in parallel computation has been motivated through the VLSI circuits where large systems of processing elements cooperate in the execution of parallel algorithms. The difficulty of choosing a specific interconnection is frequently by-passed by assuming a model (shared-memory machine) where each pair of processors is connected (Hirschberg, D.S., 1978). Other authors, e.g. Thompson and Kung [1977] and Nassimi and Sahni [1979] have suggested that processor interconnection should be limited to planar links between topologically neighbouring cells (arrays, meshes) or hexagonal interconnection (Mead and Conway, 1980). Such designs are certainly well suited for current VLSI technology, and have been used skilfully in implementing algorithms for matrices or graph problems (Levitt and Kautz, 1972) and (Kung and Leiserson, 1978a). However, this type of interconnection is not suited for efficiently implementing algorithms for various fundamental problems such as sorting and convolution. More recently, an interconnection pattern of processing elements, the Cube-Connected Cycles (CCC) illustrated in Figure 1.14, is proposed which can be used as a general-purpose parallel processor to solve efficiently a large class of problems that include Fast Fourier transform,
sorting and permutation (Preparata and Vuillemin, 1981). Because, its
design complies with present technological constraints, the CCC can also
be used in the layout of many specialized VLSI circuits. The operation of
the CCC is based on the combination of pipelining and parallelism where the
number of connections per processor is reduced by three.

FIGURE 1.14: The cube-connection of 8 processors

For a special-purpose problem, Kung (1982) offered a new approach for
the structured designs of special-purpose devices. As the current technology
offers the possibility of producing 32-bit floating-point arithmetic chips
which perform an operation in 1μs, multi-layer circuit boards holding 200
chips apiece and arithmetic units containing 10 circuit boards, therefore,
Kung's proposed machine can perform 2000 operations in 1μs at a rate of
2000 million floating-point operations per second. This operation rate is
however far in excess of that achievable by the fastest currently available
machines, e.g. array processors, CDC STAR-100.

In fact, the system discussed by Kung has two bottlenecks. The first
one is the decomposition problem which can be reduced if the algorithm
designer creates, discovers and expresses parallelism in problem formulation,
algorithm design and program development. However parallelism is crucial to
the performance of a VLSI system but the potential power of VLSI has to come
from the large amount of concurrency that it may support. The second problem
that limits the speed is the communication problem. The bottleneck for
communications appears at all levels of the memory hierarchy of a computer system. For example, at the inter-chip communication level, we observe pin bandwidth limitation. Figure 1.15 illustrates that because of pin limitation, increasing chip density does not help to reduce the number of chips needed to cover a network.

\[ \text{FIGURE 1.15: Covering a two-dimensional grid by chips with a bounded number of pins} \]

Therefore, networks that can be decomposed by removing only a small number of links are likely to be able to take advantage of the increase in chip capacity. The solution to the communication bottleneck is to design circuitry that can be partitioned among separate chips so that inter-chip communication is either completely avoided or reduced to a minimum. However, inside a chip the wiring for communication can easily dominate the power, chip area and the time required to perform a computation.

To solve the above problems, Kung [1982] has introduced the 'systolic array' in which the wiring in chip design is reduced to the minimum.

A systolic system comprises identical simple cells that circulate data in a regular fashion. The array structure for such a system provides a neat and a regular pattern that allows easy communication between cells. The information in a systolic array flows between the cells in a pipeline fashion. Figure 1.16 illustrates a systolic array composed of many arithmetic logic units (ALUs). Thus, a much higher computation throughput
can be achieved without increasing the memory bandwidth. Generally, the systolic array approach can be applied to all levels of a computer system.

In practice, many systolic arrays have been designed to solve problems in many areas such as matrix computation (Kung and Leiserson, 1978b), database manipulation (Kung and Lehman, 1980), and signal and image processing (Foster and Kung, 1980). In the design of the systolic arrays, the following ideas should be considered:

- the design should use a few types of simple cells
- data and control flow should be simple and regular
- there should be an extensive pipelining and parallelism where several data streams move at a constant rate over fixed paths in the network, interacting at cells where they meet. In this way a large number of cells are active at one time so a high computation rate can be achieved.
- multiple use should be made of each input data item. Therefore, high computation throughput can be achieved without requiring high bandwidth between the array and the memory.

In the implementation of a systolic array, each word processor can be partitioned into bit processors to achieve modularity at the bit-level. By considering the bit-serial implementation, it is possible however to implement about ten basic cells, performing 32-bit fixed-point operations on a single chip with present metal-oxide-semiconductor (MOS) technology. When many such
chips are used the systolic array can achieve a high computation rate. This is because each word is used many times in the array.

In conclusion, VLSI technology offers a great reliability at the circuit level. Also, it has the advantages of simple and regular interconnections that lead to cheap implementation and high densities where high density implies both high performance and low overhead for support components.

To this end, we completed the survey of some current parallel computer architectures where great advantages of computers of the future which are embedded in the VLSI technology is much more optimistic.
CHAPTER 2

PRINCIPLES OF PARALLEL PROGRAMMING AND LANGUAGES
2.1 INTRODUCTION

All the parallel systems discussed in Chapter 1 are powerful in the sense that a considerable throughput and speed can be achieved when they are applied to solve large problems. To increase the processing capability of these systems two different ways should be considered: firstly, the use of the primitive level hardware interface, often referred to as microcode, to provide a virtual machine capable of executing a specific high level language (Iliffe, 1982) and secondly, the exploitation of parallelism (Lorin, 1972).

Many parallel programming languages have been proposed for different types of machines. For example, Coulouris (Iverson, 1962) studied the implementation of a high level language machine for the array processing language APL. Another language such as TRANQUIL (Abel, et al, 1969), an ALGOL-like language and ACTUS (Perrot, 1980) have been designed to exploit parallelism in algorithms to be implemented in array processors such as ILLIAC IV. Subsequent work centred on the ICL-DAP (Flanders, et al, 1977) allows access to the primitive machine interface and embodies a high degree of parallelism. Finally, Flanders in conjunction with the ICL DAP has developed the DAP-Fortran (Flanders, 1982).

By developing such languages, the parallel processing environment becomes a research area of a great interest, where parallel processing is used to indicate the execution of a program comprising many "processes" (or tasks) which are processed at the same time on different processors, see Figure 2.1.

A "process" is a sequential program (i.e. a set of variables and a sequentially executed list of expressions or statements) which is controlled by at most one processor which is assigned by the operating system. A process can be executed concurrently with other processes,
delaying (at least logically) only when it needs to wait to interact with one or more other processes. Processes range from the very large, (a compiler executed under the control of an operating system) to the very small, (an expression evaluated directly by hardware).

In Figure 2.1(b), \( T_1 \) and \( T_2 \) can be executed in parallel if they are independent from each other only when both of \( T_1 \) and \( T_2 \) are completed does \( T_3 \) commence its execution.

![Diagram](image)

**FIGURE 2.1:** Sequential and parallel execution of two tasks \( T_1 \) and \( T_2 \)

The close relationship between parallel processing and time-sharing has been pointed out by Gill [1958]. Here a program is time shared on a single processor. This is useful when some parts of a program must await a signal from slower devices. Rather than having the processor remain idle, it could execute some later portion of the program. When the signal is finally received, it could then revert back to its original computations. Therefore, control can be made to oscillate back and forth between various parts of a single program.

Within an individual program, parallelism can exist at several levels:

1. Machine level, e.g. within micro-instructions
2. Instruction level, e.g. within expressions
3. Block level, e.g. between group of statements
4. Program level, e.g. between groups of programs

The program level is known as intraprogram parallelism. This later term
refers to the type of processing in which independent programs are processed concurrently. Thus, it is intended to be synonymous with the conventional meaning of multiprocessing as it is discussed in detail by Enslow [1977]. Therefore, we can state the main difference between multiprocessing and multiprogramming. A multiprogramming situation is one in which one computing element is used to work on more than one problem at one time; that is, the element is time-shared by two or more programs. Whereas, multiprocessing occurs in a situation where more than one computing element is used to work on the same problem at the same time, as described in Chapter 1, Section 1.4. In this chapter we will concentrate on the intraprogram parallelism.

There are two types of intraprogram parallelism; global and local. In the global type, a program is partitioned into tasks that can be performed in parallel. This means, whenever there are, for example, two expressions or statements in the program that have independent input sets, then they can theoretically be executed in parallel. The local type on the other hand, occurs when a single statement can be partitioned into two or more sub-statements that can be executed in parallel if they are independent.
2.2 PARALLEL PROGRAMMING CONCEPTS

Since the introduction of parallel systems, two schemes of parallelism have been investigated; implicit and explicit parallelism. In implicit parallelism, the possibilities of parallel processing are automatically detected in the program. For instance, a sequential program may be divided into parts of codes in the compilation process. Detection of the relationships between these parts allows the program to be run on a parallel computer.

This approach is appropriate for 'Data Flow' application and is in fact a basic principle of the VAL language which is developed by the data flow group at MIT (Ackerman and Dennis, [1979]). A consequence to this approach is that VAL allows programmers to only write expressions and functions; statements and procedures could have side effects that result in interference. In order to know for certain that two functions or expressions can be evaluated concurrently, a compiler for such language needs a simple rule, such that no side effects that precludes interference. Several other Data Flow languages have been proposed, such as the language LAPSE which has been developed at Manchester University (Glauert, [1978]).

Unfortunately, this approach is too restrictive for system's programs such as operating systems or data bases because, the shared information must be maintained. Another approach is when the parallelism in the program is given explicitly. In explicit parallelism, the programmer has to specify those tasks of the program that can be executed in parallel by means of special statements added to the programming language itself.

Each approach has some advantages and disadvantages. In explicit parallelism, the programmer can change the structure of an algorithm if
it is not suitable for parallel processing. However the insertion of explicit parallel programming constructs can be a time consuming and difficult process to implement and one in which mistakes can easily be made. While the programmer is not bound to recode existing programs which possess inherent parallelism for these properties can be detected using implicit recognition techniques. Therefore sequential programs need not be rewritten to run efficiently on a parallel computer.

2.2.1 Explicit Parallelism

Explicit parallelism has been studied extensively. In fact, several issues must be considered with respect to parallel processes: declaration, activation, termination, synchronisation and communication, of which the latter two are the most important. Care must be given to a special case of synchronisation, namely mutual exclusion. A quick review of past primitives can show their inadequacies. For instance, semaphores (Dijkstra, 1968), signal, path expressions (Campbell and Habermann, 1974) are clearly concerned with synchronisation. Messages (Brinch Hansen, 1970) are more oriented towards communication. Critical regions and monitors (Hoare, 1974) are especially adapted to mutual exclusion. These are described later in this section.

For expressing concurrency, several mechanisms have been proposed. For example, COBEGIN (Dijkstra 1968) or PROCESS declarations (Brinch Hansen, 1975) identify the parts of a parallel program that may be executed concurrently. This makes it possible to distinguish between the shared and the local variables. Although the new ADA language contains process declarations it allows processes and procedures to be arbitrarily nested (Ichbiah, et al 1979). This gives a great deal of programming power but makes it very difficult to understand the effects of the most
complex programs since it leads to potentially more sharing and more complex execution paths. Therefore, when using the ADA language, it is imperative that the programmer carefully limits the use of complex constructs in complex combinations.

In addition within a program loop large amounts of parallel activities can be exploited (Gosden, (1966)). Such a loop could be an ALGOL 60 `for` statement which can be extended to a parallel loop construct, PARALLEL FOR. Each iteration of such a loop may be executed in parallel.

Anderson [1965] has introduced statements for parallel processing to be used in ALGOL 60 which are of the following forms:

- `FORK L_1, L_2, \ldots, L_n`;
- `label: JOIN L_1, L_2, \ldots, L_n`;
- `label: TERMINATE L_1, L_2, \ldots, L_n`;
- `OBTAIN V_1, V_2, \ldots, V_n`;
- `RELEASE V_1, V_2, \ldots, V_n`;

where each $L_i$ represents a label and $V_i$ represents a variable. We now explain the effect of each statement.

**FORK statement:**

This statement initiates a separate control for each of the $n$ segments with the labels $L_i$. Only local labels must be used in the FORK statement and their scope is defined as the block scope. This means that this statement can operate only within the block in which it is declared, and can initiate parallel segments only at the next lower level as in general in ALGOL 60.

**JOIN statement:**

This statement is associated with the FORK statement and must occur in the same level in the program. It terminates and joins the parallel paths that we have FORKed to and a single path may follow. This action is
implemented by compiling a code that causes test bits to be available that allow the n paths to synchronise themselves after they are finished. If not all the n paths of the FORK statement have been terminated then the path following the JOIN statement cannot be executed.

An idea that follows immediately from the FORK and JOIN statements is that of co-routines. Here, we allow the FORK and JOIN type of control, but we may allow the independently executable routines to communicate with each other during their execution. This simply requires that at certain key points within the co-routines, re-synchronisation is necessary to ensure that the computed values are properly passed from one to the other. The label in the JOIN statement is the operand of the last goto statement appearing in each segment generated by the FORK statement.

TERMINATE statement:

This statement is used to de-activate program paths when some statements are not required to be explicitly defined. Since the FORK/JOIN concept is similar to the recursive subroutine calls which are already present in ALGOL 60, therefore, the recursive call mechanism relies upon finding a condition that permits a normal exit from this subroutine. Thus, the TERMINATE statement is provided to de-activate this particular call.

OBTAIN statement:

The effect of this statement is to obtain an exclusive use of the variables V_i by only one parallel path from the whole program. Therefore, the other parallel paths should be locked out so that to avoid mutual interference which may result in an undesirable effect. If this statement occurs in a block then the variables V_1, ..., V_n should be the same variables occurring in higher level blocks.

RELEASE statement:

This statement is implemented with the OBTAIN statement. It releases
those variables \((V_i)\) that have been locked out by an OBTAIN statement.

The OBTAIN/RELEASE concept is similar to the LOCK/UNLOCK concept proposed by Dennis and Van Horn, [1966]. They claimed that the execution time of a typical update sequence of data is quite small and the chance that a process will hang up on a lock instruction will be very low. It is highly possible that a process may be removed from execution if a processor is pre-empted by a higher priority computation. Therefore, a data object could remain locked for a long time if such pre-emption occurred between a LOCK/UNLOCK statement. Thus, the other processes interrogating that data may hang up for a substantial time. A solution to this problem is to inhibit interruption of a process between the execution of the LOCK and the UNLOCK statements.

Obviously, the above-mentioned statements are directed to the run-time operating system and supply enough information to control parallel and multiprogramming activities. In general, when a FORK statement is encountered, the compiler generates a code to enter the run-time executive routine and hence creates many parallel paths each of them commencing with a label of the label list of the FORK statement. Each path is assigned to any available processor and usually the first path is carried out by the same processor that carries the FORK statement itself. In the case of the number of paths being greater than the number of processors, or where no processors are available to be assigned to a particular path, then such a path is placed in a resources queue to wait for any processor to be free.

The labels contained in the label list of the FORK statement are arranged on a special forward reference list since they can be presented anywhere in the program. Therefore, on encountering a label, this list is searched and when the label is found, it is removed from this list and a special heading information is generated just before the labelled block.
This information may include code length, data etc., and it may be fed in when the program segment is completely compiled.

On the other hand, the JOIN and TERMINATE statements can be compiled as a counter code that is 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 with zero. Unless the counter is zero, the processor keeps on executing the paths which are kept in a queue waiting for execution. On the completion of executing each path, the path is terminated and released so that the processor is ready for the next path in the queue. When the counter becomes zero, the parallel paths are terminated and the processor commences the execution of the next segment of the program that starts at the first statement following the JOIN statement.

The OBTAIN and RELEASE statements are more difficult to implement because they occur just before the use of a variable. The exclusive use of variables and arrays may be implemented by indirect addressing, or by executing a data-fetch function that will determine the status of the requested data before performing the request. Two possibilities exist for the case that the request of data cannot be performed. The path executed by the processor may be suspended awaiting access and the processor reassigned to other work; or the processor can be idle by continuously attempting to access data until the data in the question is released by a RELEASE statement.

The OBTAIN/RELEASE concept is one approach which is implemented to solve the synchronisation problem. First, the notion of semaphores was suggested by Dijkstra [1968] to prove many synchronisation properties. Semaphores alone do not reduce the need for large numbers of shared variables, so there is still a large potential for interference. However, they have been successfully used for a harmonious cooperation of a system
of several processes. Following the introduction of the semaphores the conditional critical regions was developed (Brinch Hansen, 1973) and (Hoare, 1972). Critical regions group shared variables into resources, allow exclusive access to them, and allow invariants on shared variables to be established on a per region basis. Therefore, the conditional critical regions reduce potential interference and aid to the establishment of invariants by grouping shared variables into disjoint sets dependent on their use.

In order to further reduce the interference among processes, monitors have been introduced so that the shared variables are grouped together as well as the statements that access them (Brinch Hansen, 1973 and 1975). An alternative approach to monitors is to prohibit shared variables and instead structure a system as a collection of processes that protect their local variables and communicate by exchanging messages. It has been proved that the correctness of a system of processes depends only on their individual correctness and their correct use of the shared message buffers. In his paper, Hoare [1978] has suggested that parallel composition and communication of processes should be accepted as a primitive concept in programming. In his recent paper, Hoare [1980] has introduced a simple mathematical model for communicating sequential processes, where the proof of correctness of programs expressed as communicating sequential processes is provided.

From the basis of the concept of communicating sequential processes, the implementation of mutual exclusion of several asynchronous parallel processes is developed (Burns, et al, 1982). In fact, concurrent processing by several asynchronous parallel processes differs from sequential processing in that the order in which the steps of the processes which are executed is not predetermined but depends on difficult-to-predict variables such as the
relative speeds of the processes and some external events such as interrupts and operator intervention. To prevent interference among the processes, a critical section is designated in each process to include the section of code that every process wants to execute perhaps at the same time. However, the critical section cannot be executed simultaneously by two or more processes. As described earlier the mutual exclusion of the access to critical sections is provided by means of entry and exit protocols to these critical sections. However, it is the job of the protocols to ensure that only one process at a time can access the critical section while the other processes trying to access it must wait. Also, the protocols play a scheduling role in determining which of the several contending processes is allowed to proceed. A particular study of the cooperative processes with critical regions in a MIMD machine is described in Section 2.3.

To illustrate the effect of the FORK/JOIN concept we present the following example (Figure 2.2).

Program 1

```
FORK L1,L2
L1 DO 10 I=1,N+1
10 A(I)=2*I+3
   GOTO L8
L2 DO 20 I=1,N+1
20 READ B(I)
   DO 30 I=1,N
30 C(I)=B(I)*B(I+1)
L8 JOIN L1,L2
   FORK L3,L4
L3 DO 40 I=1,N
40 D(I)=C(I)**2/A(I)
   DO 50 I=1,N
50 WRITE D(I)
   GOTO L10
L4 DO 60 I=1,N
60 WRITE C(I)
L10 JOIN L3,L4
```

FIGURE 2.2: A simple program using the FORK/JOIN statements

The statement FORK L1,L2 generates two paths; one initialises the
array A and the other reads B and computes C, which can be carried out simultaneously on two processors. The execution will be carried out to the segment following the FORK L1,L2 when both the paths labelled L1 and L2 have encountered the JOIN statement. This is implemented by a compiling code that allows the two paths to synchronise themselves after they are finished. When FORK L3,L4 is encountered, the D array is computed depending on the computed arrays A and C from the previous paths and is printed out while the second path writes out the array C computed previously. The execution of this program leads to the flowchart in Figure 2.3.

![Flowchart of Program 1](image)

**FIGURE 2.3:** The flowchart of Program 1

The dotted lines in Figure 2.3 show the data dependencies in this program. This communication of the paths or so called co-routines requires special
attention in some "critical points" in the coroutines. Therefore, re-
synchronisation is necessary to ensure that the computed values are properly
passed from one path to the other. However, we try to program the parallel
machine with any number of paths as independent as much as possible to avoid
the above situation.

2.2.2 Parallel Computational Models

The general topic of parallel processing refers not only to the parallel
machines and the software systems which operate on them, but also to the
organisation of computations which are to be executed in parallel. However,
numerous models for describing parallelism in computer systems have been
proposed in the last fifteen years. Some models (schemes) are aimed mostly
at formal descriptions and hence suitable for correctness analysis (Miller,
1973). Examples of such models are, Petri nets, computation graphs, and
parallel program schemata which we describe in this section.

In the theoretical investigation of properties of parallel processes
two different approaches appear. First approach is to restrict the basic
local entities of the models and their interconnection so that the properties
desired, such as determinacy, can be shown to hold. This approach allows
interconnection of units according to specified rules of interconnection
between adjacent units so that the desired properties hold (Adams, 1970).
In this paper, Adams presented a directed graph as a model for parallel
computations in which the sequencing control is governed by the flow of
data. All programs expressed in this model are determinate, that is, the
results of the computation do not depend on the number of processors used,
their relative speeds, or the possible sequencing of computation steps that
are performed. However, a parallel processor system designed to execute
graph programs would ensure that the results of the computation would be
the same regardless of the system configuration. Such a system would allow processing resources to be dynamically allocated to the computation. This could be governed either by the demand for resources by the computation or by the availability of resources within the system.

In the second approach, a model with less constraints may be used for the process, but necessary and sufficient conditions for the properties of the model to hold should be established. This approach is used in the parallel program schemata studies (Brinsfield and Miller, 1971), and (Karp and Miller, 1969). However, these two approaches prepare a good study to understand the behaviour of some properties.

Petri Nets

Petri nets were originally introduced by Petri [1966] and have been further developed and studied as a method of representing processes (Commoner, et al, 1971).

A Petri net is represented as a graph with directed edges between two different sets of nodes, a set of places corresponding to conditions which may hold in the system, and a set of transitions representing events which may occur. The places in a Petri net may contain tokens which signify the holding of the corresponding conditions. For a given transition, those places that have edges directed into it are called input places and those places having edges directed out are called output places. If all the input places for a transition contain a token, then the transition is said to be active. Graphically, places are denoted by circles; transitions by bars; and tokens by dots inside a circle. A transition may fire when all its input places are full (i.e. contains a token). After a firing, a token is removed from every input place and a token is added to every output place. Thus, a token in a place can be used in the firing of only one transition.
The firing of a transition corresponds to the occurrence of an event. Sequences of transition firings are called execution sequences. An execution sequence represents a simulation of the system being modelled with the initial markings representing the initial conditions.

A simple example of a Petri net is shown in Figure 2.4, where the starting condition has a token only in place $P_1$. The activity of the net (or process) is described by the successive firings of transitions. In this figure, $t_1$ can fire followed by $t_2$ and $t_3$. After both $t_2$ and $t_3$ have fired, $t_4$ and $t_5$ can be active where either $t_4$ or $t_5$ can fire but not both. After all the firings are finished, the net is back to its starting condition and the process is ready to repeat. Since more than one edge can leave and enter a transition or place, therefore, the rule that a token in a place can be used in the firing of only one transition affects how a token can be used when the place has edges going to more than one transition. This explains why either $t_4$ or $t_5$ fires and not both. Thus, $t_4$ and $t_5$ are said to be in conflict since they share a common input place.

[Diagram of Petri net showing transitions $t_1$, $t_2$, $t_3$, $t_4$, and $t_5$ with places $P_1$, $P_2$, $P_3$, $P_4$, and $P_5$.]

FIGURE 2.4: A simple Petri Net.

It is obvious from the above description that Petri nets can conveniently describe concurrency of events. Also, by allowing more than one token in a place, a backlog of tokens can be built up which could be used by later firings.
Various types of restricted Petri nets have been studied extensively to represent the control of concurrent processes (Commoner, et al., 1971, and Izbicki, 1971). For example, a marked graph is a sub-class of Petri nets in which every place has exactly one transition entering and one transition leaving. By interpreting the places as edges directed from entering to leaving transitions, and transitions as nodes, transforms a marked graph into a directed graph. Therefore, tokens appear as "markings" on the edges of the directed graph.

Petri nets can represent programming language constructs where a transition is represented on each arc of a flowchart and the nodes of the flowchart represent the places. In this case, we obtain a Petri net representing a program where the transitions will correspond to executable statements of the program and the places will also act as deciders, besides having their usual meaning of condition holders. In Figure 2.5, a Petri net presenting an IF-THEN-ELSE construct is illustrated.

![Petri net representation of an IF-THEN-ELSE construct.](image)

In addition, Petri nets may represent the partitioning of tasks.

For example, the following ALGOL-like program:

```plaintext
PARBEGIN
S1: PARBEGIN
   S1.1: BEGIN ... END;
   S1.2: BEGIN ... END;
   PAREND;
S2: BEGIN ... END;
S3: BEGIN ... END;
PAREND
```
is illustrated as a Petri net in Figure 2.6. On the other hand, Petri nets are extremely valuable for representing process synchronisation and mutual exclusion. For example, a semaphore can be represented as an input place shared by the transitions (critical sections) which are to be mutually exclusive. This approach is illustrated in Figure 2.7.

**FIGURE 2.6:** A parallelism model by Petri nets

**FIGURE 2.7:** Mutual exclusion in Petri net representation

Petri nets have a major limitation in their modelling ability, i.e., their incapacity to count the number of tokens on a place or to test whether a place is empty. This lack of descriptive power can be extended by the firing rule. Baer, [1982] has further described Petri nets and their extensions in a graph model where the control and data flows expressing the
concepts of parallelism which include partitioning, pipelining and the process synchronisations are all successfully implemented.

**Computation Graphs**

Karp and Miller [1966] presented a theoretical graph model of computations to express parallelism in simple repetitive processes such as those occurring in the "inner loops" of computations. In this model, which is a labelled directed graph, each node of the graph corresponds to an operation in the computation, and each edge represents a queue of data, assembled in a first-in-first-out fashion, directed from one node to another. The performance of an operation takes operands off the incoming edges and places results on outgoing edges. The initialisation of a computation step is determined by the lengths of the queues on the edges directed into its associated node. However, an operation can fire only if there are sufficient operands on the incoming edges. Any edge may require many operands, as well as several results may be placed on an edge. This type of model has restrictions on the type of the control sequence which can be represented within the model which arise from a fixed queue discipline, that is, each data queue has a unique 'source' and a unique 'sink' and the data dependent conditional transfers cannot be represented.

The model that Karp and Miller, [1966] have designed has the following properties:

1. To prove determinacy, i.e., for a given input, the program will yield a unique output independent of the relative processor speeds.
2. To test whether a given computation will terminate;
3. It supplies a procedure for finding the number of performances of each computation step; and finally,
4. The amount of temporary storage required for the data queues
associated with edges of the graph together with conditions for the queue lengths to remain bounded.

A simple example of a computation graph is shown in Figure 2.8 for computing the Fibonacci numbers of the form,

\[ a_k = a_{k-1} + a_{k-2}, \]

where \( k = 2, 3, \ldots \), and \( a_k \) is a natural number.

FIGURE 2.8: A simple computation graph

The example in Figure 2.8 clarifies how the computation graphs are convenient for representing many repetitive processes.

The computation graph is different from a Petri net in several respects. First, each queue of data is associated with only two operations, one putting results in the queue, and one taking operands from the queue. However, in Petri nets, sharing is permissible by allowing a place to have edges directed to more than one transition. A second difference is that in Petri nets a place may contain more than one token, but no discipline on the queues of data are imposed in the model, whereas the computation graph represents the data items queued in a particular manner.

Finally, the computation graph has some limitations that make the model inappropriate for representing a variety of computations. These limitations are: first, no common memory for data; and second, the lack of a facility for conditional branching. Although some considerable work has been achieved on the graphical models of computation, it has been found that several useful approaches for representing parallel processes that ensure determinism have been completed. Also an appropriate scheme of interconnecting the nodes has been implemented successfully. However, as
an extension to these models together with the study of how to overcome the above-mentioned two limitations have led to the formulation of more general schemata models which are termed the parallel program schemata.

**Parallel Program Schemata**

In Karp and Miller, [1969], a parallel program schemata has been described. Let $\delta=(M,A,J)$ be a parallel program schemata which consists of: a set $M$ of memory locations; a finite set $A$ of operations; and $J$ a control for sequencing operations. For each operation $a \in A$ there are a set of domain locations $D(a) \subseteq M$ and a set of range locations $R(a) \subseteq M$ associated with it. Thus, when an operation $a$ is initiated it obtains its operands from locations $D(a)$, and when it completes its performance it places its results in the $R(a)$ locations. The control $J$ is formulated as a transition system composed of a set of states and a particular partial transition function.

The control of the schema has the following properties:

1. The control of the schema must allow the initiations and terminations to occur, starting from the starting state and proceeding step by step. Also, an operation termination can occur only if it has previously been initiated;

2. For a finite computation all previously initiated performances must have terminated;

3. The third property is the "finite-delay property". Once an operation is initiated its termination must occur after only a finite number of other initiations and terminations. That is, the computation being performed by the operation does not require an infinite amount of time. It is known that one problem in parallelism is to determine how much parallelism is possible in a process. For the schemata respect, this
translates into finding a schema that is equivalent to a given schema and also maximally parallel. However, many studies develop a sequence of transformations on the schemata that increase parallelism and develop a language for expressing and studying the structure of computational processes.

In the last few years, work has been completed on program transformations to allow traditional programming languages to be implemented on different parallel architectures. Kuck [1978] and [1981] has studied these concepts extensively. He presented a directed graph to be associated with a program. To illustrate this, first consider two assignment statements $S_1$ and $S_2$.

Suppose that the assignee of $S_1$ is going to be used, after $S_1$ execution, in the Right Hand Side (RHS) of $S_2$ then we say that $S_2$ is data dependent on $S_1$. On the other hand, if the assignee of $S_1$ is going to be used in the RHS of $S_2$ without considering its obtainable value from executing $S_1$ then we say that $S_1$ is data antidependent on $S_2$. Finally, if the assignee of both $S_1$ and $S_2$ denotes the same variable and that the assignee of $S_2$ is stored after that of $S_1$, then we say that $S_2$ is data output dependent on $S_1$.

Consider the following example:

$$
S_1 \quad A + B + C \\
S_2 \quad A + C * 3 \\
S_3 \quad C + A + C,
$$

this example represents a block of assignment statements and assumes the execution starts with $S_1$, then the data dependent graph will be:

- $S_1$ data output dependent on $S_2$.
- $S_2$ data antidependent on $S_1$.
- $S_1$ data output dependent on $S_3$.

where $\rightarrow$ means data output dependent and $\leftarrow$ means data antidependent and $\longrightarrow$ means data dependent.
According to these properties the block, which is a collection of assignment statements, can be classified to distinct forms according to the execution of these statements. The form could be simultaneous (SIM), sequential (SEQ), together (TOG), fetch-store (FS) or store-fetch (SF). These are described in Kuck [1978].

If several processors share a memory, a TOG block can be used to indicate that any processor may be used to execute any assignment statement in any order. For example, in the block,

\[
\text{TOG}[A + B + 3; C + D \times E; F + G \times 2/H]
\]

three processors can execute these three statements simultaneously. Of course, FORK and JOIN statements are the statements which can give the parallel paths for these sort of blocks. For a single assignment statement containing an expression of \(n\) atoms, Kuck [1978] has notified that by various tree-height reduction methods the expression can be evaluated in \(O(\log n)\) steps. More generally, a program may contain a loop. A loop may be acyclic, linear cyclic, or nonlinear cyclic. A cycle may involve one or more statements. If the RHS are linear combinations of the LHS variables, it is called a linear cycle (Kuck, [1978]). A loop containing a linear cycle is called a linear cyclic loop. If a cycle is not linear, the loop containing it is a non-linear cyclic loop. If a loop contains no cycles, it is called an acyclic loop. For any type of loops in a program, an efficient technique may be used to transform programs written in an ordinary sequential programming language into programs ready to be executed on different machine structures. For example, Kuck [1981] implemented these techniques on array and pipeline machines. His approach can well be used to determine cost-effective architectures for given computer system job loads.
2.2.3 Implicit Parallelism

This approach of parallelism is associated with sophisticated compiling and supervisory programs and the related overheads. Its effectiveness lies in that it is independent of the programmer, and existing programs need not be modified to take advantage of inherent parallelism. However, in implicit parallelism, it will be necessary to analyse the program to see how it can be divided into tasks. Therefore, the compiler could detect parallel relationships between tasks as well as carrying out the normal compiling work for a program to be run on a serial computer.

Different methods have been developed that possess the feasibility of automatically recognising parallelism in computer programs. For example, Bernstein [1966] presented a method based on set theory. His work is mainly based on the four ways in which a memory location may be used by a set of instructions or sub-program $P_i$.

These are:

1. The location is only fetched during the execution of $P_i$.
2. The location is only stored during the execution of $P_i$.
3. The first operation involving this location is a fetch. One of the succeeding operations of $P_i$ stores in this location.
4. The first operation involving this location is a store. One of the succeeding operations of $P_i$ fetches from this location.

Several relationships exist between any two sub-programs (Bernstein, 1966). Following this work, Evans and Williams [1978] have described a method of locating parallelism within ALGOL-type programming languages, where some constructs such as loops, if statements, and Assignment statements are studied. For implementing these constructs Williams [1978] has presented an ALGOL 68-R program that describes how an existing multi-pass compiler can
detect potential parallelism. This compiler is extended by adding two
stages to it, the 'Analyser' and the 'Detector'. The analyser divides a
given ALGOL-type language into sub-programs called stanzas. The analyser
limits the size of a sub-program to be a specific program construct (e.g.
a loop) or a collection of statements using not more than fifteen different
variables. The detector, on the other hand, will take the sub-programs
produced by the analyser and carry out the tests to determine which parallel
relations, if any exists between these sub-programs. The detector program
examines the relationship between pairs of sub-programs or within a loop
for a parallel computer system with shared memory and with or without
individual private memories. With the information from the analyser and
detector programs it is possible when compiling a serial program to identify
parts of programs that may be run in parallel.

Another research topic is included in the study of implicit parallelism.
This is the parsing of arithmetic expressions which are executed on a
parallel computer with a number of arithmetic units or processors (Gonzalez
and Ramamoorthy, 1970) and (Williams, 1978). It was shown that the time
taken for an arithmetic expression to be calculated on a parallel computer
can be estimated to be proportional to the number of levels in the tree
representation of the expression. Whereas, for a serial computer the time
taken to calculate an expression can be estimated to be proportional to the
number of operations needed to be performed.

In any parallel computer, if there are sufficient processors then any
operations that appear at the same level in a tree representation of an
expression may be executed in parallel on separate processors. Now,
consider the expression,

\[ A + B + C + D + E + F + G + H \]

which is represented in Figure 2.9(a),(b). From Figure 2.9(a) it can be
seen that this expression requires 7 units of time, whereas, in Figure 2.9(b) the same calculation of that expression only takes 3 units of time. The tree representations of this expression have seven and three levels respectively. However, in Figure 2.9(b), four processors are required at level 1, two at level 2 and one at level 3. Whereas in Figure 2.9(a) only one processor is required in every level. Therefore, we can say that the tree representation of Figure 2.9(b) shows there is more potential parallelism than in the tree representation given in Figure 2.9(a). We conclude that, in a parallel processing environment, the amount of potential parallelism for the execution of an expression is inversely proportional to the number of levels (or height) of the tree representation of that expression. Therefore, a tree representation of an expression should be formed with the least possible number of levels to achieve higher order of parallelism.

FIGURE 2.9: Possible binary tree representations of the expression $A + B + C + D + E + F + G + H$

Many algorithms have previously been proposed for recognising parallelism at the expression level. These are Squire's, Hellerman's,
and Baer and Bovet's algorithms. Descriptions of these algorithms are found in the reference (Williams, 1978). Recently, Kuck [1978] examines the application of distribution over expressions such that a tree representation is of minimum height. This may involve performing extra operations such as shown in Figure 2.10. The distribution form (b) requires five operations whereas the form in (a) only requires four operations. However, (b) is completed in three levels whilst (a) requires four levels.

![Figure 2.10: Possible tree representations of A * (B * C * D + E)](image)

Also, Williams [1978] has developed a new algorithm for parsing expressions and found a new technique to produce a binary tree representation of arithmetic expressions. Some constraints are applied to her algorithm, and these are:

1. The priority of brackets are observed.
2. Expressions are not to be re-ordered.
3. The tree representation should be of minimum possible height.

Amongst all the algorithms previously mentioned, Williams' algorithm is shown that it presents its results in a form suitable for determining the maximum amount of parallelism without unnecessarily affecting sensitive numeric equations (Williams', 1978).

To conclude, explicit and implicit parallelism are studied and, in fact, they can be considered when any new parallel machine is built, especially explicit parallelism which has basic concepts for most parallel machines, but may vary in some specific concepts when implemented on different machines.
2.3 PROGRAMMING THE NEPTUNE SYSTEM

The hardware configuration and the operating system of the NEPTUNE MIMD system was described in Chapter 1. The programming aspects of this system will now be described in this section. The programming language available on the NEPTUNE system is FORTRAN, extended by adding several programming constructs, i.e. macros to achieve the parallel processing requirements.

A parallel program, which has the form illustrated in Figure 2.1 above, possesses the following properties:

(1) Each path (task) must be executed by only one processor while at the same time other processors must be informed and locked out of that path. Without this mechanism, stability of the results cannot be assured.

(2) Only when all parallel paths have been completed a given path following them can commence execution. When this later path makes use of variables which are set by the preceding parallel paths, they must then be made available to the processor carrying out this path.

In order to program such conceptual structures, the programmer should define where the parallel paths start and end, which data is to be shared between the paths, and synchronisations to ensure the reliable update of certain shared data structures. First, we present some necessary constructs which are essential in any parallel program implemented on the NEPTUNE system.

(1) $USEPAR

This construct must be the first executable parallel statement. On encountering the $USEPAR all processors except one are forced to wait until parallel paths are created for them to execute.
(2) $END

This statement replaces the END statement of FORTRAN. It forces the checking at pre-compile time that the nesting of parallel syntactical constructs is completed within each individual subroutine.

(3) $STOP

This replaces the STOP statement of FORTRAN. It ensures the termination of the program.

Within each parallel program there is a piece of code that can be evaluated in parallel. Therefore, there are three pairs of constructs available on the NEPTUNE system to generate and terminate such codes (or paths). They are as follows:

$DOPAR 1 I=N1,N2,N3
    ... 
    'code using index I'
    ...
1     $PAREND

This construct is similar to the FORTRAN DO loop but it generates \((N2-N1+1)/N3\) paths with identical code and each path has a unique value of the loop index \(I\) (i.e. \(N1, N1+N3, \ldots, \text{etc.}\)). Thus, by organising a formula for indexing the data by the variable \(I\), it ensures that different paths evaluate different results. $PAREND is similar to the CONTINUE statement in FORTRAN where every path generated should be terminated once it encounters $PAREND.

On the other hand, to generate paths with different code, we use the construct FORK/JOIN:

$FORK 1,2,3;100
1   'code for path 1'
   GOTO 100
2   'code for path 2'
   GOTO 100
3   'code for path 3'
100  $JOIN
The construct is equivalent to the computed GOTO statement in FORTRAN. In this construct, the paths commence at labels 1, 2, 3 and each path except the last path contains a GOTO statement at the end to force each path to terminate at the label 100 of the JOIN statement.

Finally, to generate paths with the same code where each processor is forced to execute the code once and once only, we use,

```fortran
$DOALL 100
  'code'
100 $SPAREND
```

This construct is used to initialise the data or to obtain the timing information because only the previous two constructs allow any processor to start execution of a path.

In all the three constructs above, nesting is allowed but considerable care should be taken to handle the variables being used in the inner loops. However, the local variables of a parent path are not made available to the children. Note that all the index values of the ancestors of a path, together with the index value of the path itself, are restored prior to the execution of a path. Figure 2.11 illustrates an example of nested parallelism which can be implemented on the NEPTUNE system.

![Diagram of nested parallelism]

**FIGURE 2.11: A possible nested parallel construct**
The two constructs, $FORK/$JOIN and $DOPAR/$PAREND are implemented in a similar manner. The implementation was first completed on the Interdata Dual system (Barlow et al, 1977) and (Slade, 1982) at Loughborough University. This implementation is the same, perhaps with some minor modifications to the NEPTUNE system. However, instead of viewing the problem as one task with several processors, a more general approach is to consider the parallel job as one which creates other tasks during its execution.

All the tasks in the program are scheduled for execution on one of the processors, thus we have a dynamic scheduling list accessible to all processors. A scheduling algorithm was devised to arrange the distribution of the tasks between the processors. The scheduler maintains a block of information, called a task control block (TCB), for each task known to it. Sufficient information is contained in a TCB such as the start address of each path, some codification of the precedence relations controlling when the path can be executed, and whether the path has to be executed by a particular processor.

When any parallel construct is encountered (e.g. $FORK$, $JOIN$, $DOPAR$, $PAREND$), the scheduler is called by the processors to add or delete the TCBs from the scheduling list. The processor that executes the $FORK$ or $DOPAR$ instruction takes the first path from the list and executes it, followed by the other processors that execute some other tasks. If no task is found the processor can be switched to processing other work using normal multiprogramming techniques. The instructions $JOIN$ and $PAREND$ are counters which are set initially to the number of parallel paths decremented each time a path is encountered. If the execution of a path is completed, the processor that carried it can be reassigned to the next path in the list.
The parallel paths have precedence relations. This means, once parallel paths are encountered in the $FORK$ instruction, they can be executed by any available processor. Only the path following the $JOIN$ instruction has to wait execution until this instruction has been executed by each of the paths referenced on the corresponding $FORK$ instruction. This precedence condition was handled by placing the label of the $JOIN$ instruction on the $FORK$ instruction as in our previous example of the $FORK$/JOIN construct. However, to secure the variables track and to maintain their correct setting, the processor continuing after a $JOIN$ must be the processor which issued the corresponding $FORK$. Because, the variables which are not used in the parallel paths are stored in private memory. Those private variables used in the path preceding the $FORK$ and required again in the path following the $JOIN$ but not used in the paths created by the $FORK$ instruction will have the correct values only in the copy of the program stored in the private memory of the processor that executed the path preceding the $FORK$ statement. Since the TCB for the path following the $JOIN$ is created when the corresponding $FORK$ is encountered, it is important to place the number of the processor in the TCB to enforce this processor selection.

In any MIMD machine, synchronisation (or co-ordination) between parallel paths is required. The synchronisation is of three types: (1) when parallel paths are created by $FORK$ or $DOPAR$ construct the termination of these paths requires synchronisation. (2) In the case when a path produces results that might be required by another path. Therefore, synchronisation is required to transform these results. (3) Synchronisation is required when parallel paths try to access shared resources. Thus, these shared resources must be accessed in a controlled manner to prevent them being corrupted. The resources are some shared data structures in the
user program or a shared disc drive. Therefore, synchronisation is required whenever a path wants to communicate with its environment.

It was mentioned in Section 2.2.1 that many instructions have been proposed to enable access to shared resources to be synchronised. A basic algorithm has been developed by Lamport [1974], which guarantees access to a shared resource on a first-come-first-served fashion. However, each processor that requires to access a resource has an associated numeric variable. The value of this variable indicates the order in which the processors carrying paths may access the resource. When a processor wishes to access a resource it has to assign a new value to the variable which is one more than the largest number of the other processors. This processor then inspects the number of each processor and it may use the resource if it is ranked earlier than the others. When it finishes with the resource, it assigns zero to the variable.

Another approach is a resource master technique in which one processor is responsible for the resource until the ownership is passed to another. This approach is used on the NEPTUNE system. In this method, a single location is used to store the current owner and each processor is given a unique identification number. When a processor wishes to access a resource, it raises a flag and waits until its number is placed within the owner location. In this case, the processor is considered as a master of the resource until it releases this resource where the ownership can be passed to another processor that requires it. If it happens that no other processor wants this resource, then the processor must dispose of the resource when it finishes its ownership (Woodward, 1982).

On the NEPTUNE system, many programming constructs have been implemented to maintain the use of variables which are shared amongst parallel paths and
how to access shared data structures contained in critical regions. For the former case, the construct (\texttt{\$SHARED list of variables}) is implemented. This forces the variables in the list to be loaded into the shared memory. All other data, including the program code, is held in the local memory. This construct is similar to the \texttt{COMMON} statement in FORTRAN.

In the case of critical sections, the NEPTUNE's user can use up to eight resources which can only be owned by one of the processors at any one time. These resources must be declared with FORTRAN-like names using \texttt{\$REGION list of names} the scope of this declaration being the next \texttt{\$END} construct. To claim and to release resources, two constructs have been implemented on the NEPTUNE system:

\begin{verbatim}
\$ENTER name
\$EXIT name
\end{verbatim}

the critical section is embedded with an \texttt{\$ENTER/\$EXIT} pair of constructs. The same resources can protect different critical sections in the program. A critical section can include sub-routine calls and claims for resources can be nested, but a good care should be taken to avoid the possibility of dead-lock situations. Moreover, the resources named on the region are bound to system entities at the preprocessor stage.

The other side of programming the NEPTUNE system is how to compile and run a parallel program. The command XPFCL is used to produce a load module from the user's source program. The effect of this command is to:

1. preprocess the user source program to convert the special parallel constructs into FORTRAN statements;
2. compile the resultant FORTRAN,
3. link the compiler output with the available FORTRAN libraries and machine code written routines to control the parallelism;
4. store the resultant load module in the user's program file.
If an error is detected in one of these stages, then the command will be terminated and an explanatory message about the error will be written to the file.

More recently, another command for compiling is implemented; the XPFCLD command. This command has the effect of the XPFCL command except that with this command many parallel paths (up to (2^5 - 1) paths) can be created and executed efficiently on the available 4 processors.

On the other hand, to run a parallel program on the NEPTUNE system, the XPFT command is used. The system responds to this command and the user should declare on which processor(s) this program should run, the name of the load module, and finally whether this execution is required in the foreground or on the background computing basis. Since the processors are known by their number 0, ..., 3, therefore, the number of the processor that executes the program is mentioned in the list and always processor 0 should be one of these processors, because, it includes the main disc. In the case of run-time errors which occur in the program, the command will be terminated and a message, like 'floating point overflow error', is repeated to the user.

In fact, before the execution command the user must specify which input and output files are used for the data and the results. This specification is achieved by the AS (Assign Synonyms) command by which the user should identify the channel unit from which the input data is obtained and the channel unit for the output file that includes the results.

Since the performance of parallel programs is of interest, therefore, the timing results of such a program should appear in the output file. To obtain the correct time that the path is active, the following constructs are used:

```
$DOALL 100
CALL TIMEST
100 $PAREND
```
This construct forces each processor to start timing. This construct is usually placed immediately before the main job to be performed by the path. The second construct is

$DOALL 200$
CALL TIMEOUT(ITIME)
$200$ $S$PAREN$d$

where this construct is placed after the main code in the path. However, it calls every processor to supply its timing results and places it in the ITIME array which is usually declared as a shared array of size 100 and its result should be arranged in 8 columns. The timing results for each processor are held in ITIME as follows:

ITIME(1+j*25)...ITIME(24+j*25) hold timing results for processor
j=0,1,2,3.

Let i=j*25, then we have:

ITIME(1+i) \{ clocked cpu time in seconds and milliseconds
ITIME(2+i) \{ elapsed time in seconds and milliseconds
ITIME(3+i) \{ number of parallel paths run by this processor
ITIME(4+i) \{ number of waiting cycles because no path is available
ITIME(5+i) \{ number of accesses to critical section resource 1
ITIME(6+i) \{ number of waiting cycles because this resource is being used by another processor
ITIME(7+i) \{ same for critical sections resources 2 to 8
ITIME(8+i) \{ information on system critical section resource
ITIME(9+i) \{ " " " " " " "
ITIME(22+i) \{ " " " " " " "
ITIME(23+i) \{ " " " " " " "
ITIME(24+i) \{ " " " " " " "
ITIME(25+i) \{ is not used.

Most information about the algorithm performance is obtained from the ITIME array as will be made clear later when we discuss the algorithms performance analyses in Chapters 4, 5 and 6.
CHAPTER 3

THE STRUCTURE OF PARALLEL ALGORITHMS
3.1 **INTRODUCTION**

The literature on parallel algorithms covers a wide area of research on different types of parallel computers. In fact, parallel algorithms have been studied since the early 1960's, although at that time no parallel computers had been constructed. Many researchers find designing parallel algorithms fascinating and challenging. However, increasing interests in parallel algorithms have been created by the emergence of large-scale parallel computers, i.e. DAP ILLIAC, CRAY, CYBER. Therefore, a variety of algorithms have been designed from different viewpoints and for the various parallel computer architectures which were described in Chapter 1.

Different types of parallel computers can compute algorithms designed with some sense of parallelism. The computation of parallel algorithms has to be a fully independent computation, by which we mean that the results obtained from executing one segment of the algorithm are not affected by the results of another segment. Although, this case cannot always be encountered, some algorithms may already contain independent computations that are not needed to be reorganised. Such algorithms are said to have inherent parallelism which was defined in Chapter 2.

As an example of such independent computations, consider the addition of two n-vectors $\mathbf{A}$ and $\mathbf{B}$ to yield another vector $\mathbf{C}$ such that,

$$\mathbf{C} = \mathbf{A} + \mathbf{B},$$  \hspace{1cm} (3.1.1)

where $\mathbf{A} = (a_1, a_2, \ldots, a_n)$, $\mathbf{B} = (b_1, b_2, \ldots, b_n)$ and $\mathbf{C} = (c_1, c_2, \ldots, c_n)$.

Clearly, the evaluation of the components of the resultant vector $\mathbf{C}$ are obtained from the formula,

$$c_i = a_i + b_i, \text{ for } i = 1, 2, \ldots, n.$$  \hspace{1cm} (3.1.2)

Therefore, this calculation is independent and a computer with $n$ processors can compute this vector in one step, where each component is evaluated in one processor.
As was discussed in Chapter 2, implicit parallelism is not always exhibited in the sequential algorithms, i.e. algorithms run on a single processor or sequential computer. In this case, the algorithms have to be reconstructed, using special programming constructs (see Chapter 2), to exploit parallelism. For this reason, we can conclude that a good sequential algorithm does not always lead to a good parallel algorithm.

On designing a parallel algorithm, the computer implementing this algorithm should be considered. The different computer architectures discussed in Chapter 1 execute different types of algorithms, but sometimes an algorithm can be implemented with some modifications on more than one type of computer.

Many algorithms have been designed for the SIMD computers since the late 1960's, where the area involving MIMD computers was scarcely investigated. This does not mean that the SIMD computers are superior to MIMD computers. Rather, the problems that face MIMD computers seemed to be more difficult to solve than for SIMD computers. On the other hand, the MIMD computers have been built up comprising few processors, e.g. 2, 4 and now up to 32 processors, whereas SIMD computers have an effective parallelism as small as 32 to upwards of 256 (see Chapter 1). With all these differences, researchers are interested to investigate the two different systems, each with its advantages and disadvantages to exploit the differences in parallelism. For example, Barlow et al [1982a] prepared a comparative study of solving the problem of finding the eigenvalues of a real symmetric tridiagonal matrix on pipeline, SIMD and MIMD computers.

In general, we can say that algorithms executed on a SIMD computer require a high degree of parallelism because this type possesses up to order \( n^m, O(n^m) \), processors, where \( m=2,3,\ldots \), while a MIMD computer possesses
up to $O(n)$ processors, where $n$ is the number of the subtasks of the problem. This does not mean that if an algorithm designed for MIMD computers cannot be run on the SIMD computer but, the thing is, if that algorithm contains a large number $n$ of independent computations then only $n$ of the processors of the SIMD computer will be used concurrently and the rest being unused. Conversely, the MIMD computer would not have sufficient processors to execute $O(n^m)$ independent computations at the same time but instead it groups each of $P$ computations and executes them simultaneously, where $P$ is the number of processors.

We know that the processors of a MIMD computer are asynchronous so that they need not necessarily be involved on the same problem. Thus, if a new processor is added and it has been noticed that it is of no effect on the run-time of an algorithm then it would be better to use that processor on a different problem. However, in the design of algorithms for a MIMD computer we try to efficiently use all the processors.

On the other hand, the processors of a SIMD computer are synchronous and hence they cannot be used to run independent computations that are not identical and they must remain idle when not required. However, the non-identical computations would be executed sequentially on the SIMD computer (thus reducing the efficiency of the system). Thus, on designing algorithms for a SIMD computer, one should consider only algorithms with substantial amounts of identical computations in order to achieve high efficiency.

In general, we conclude that a good MIMD algorithm is not always a good SIMD algorithm and vice versa.

Now, if we consider the pipeline computer, we see that the algorithm designed for this type of computer should be as a string of identical
operations that may be treated in an assembly line fashion. Actually, the situation is the longer the string the greater the efficiency achieved. For this reason, a good pipeline algorithm is generally a good SIMD algorithm and vice versa. This observation was made by Stone [1973a] where it is based on the fact that the SIMD and pipeline computers are heavily oriented to vector processes.

In data driven systems, an algorithm is organised so that the computation of its statements is sequenced by the availability of the data. A well-formed data flow program is defined recursively as: an operator, an acyclic graph of data flow programs, a conditional whose branches are disjoint data flow programs, or a loop whose body is also a data flow program. Usually, well-formed data flow programs are determinate and deadlock-free. In such programs, the sequence of execution of parallel operators does not affect the result because each operator executes as soon as its required operands are available, data flow notation does not impose arbitrary sequencing constraints on instruction execution.

To allow users to take fuller advantage of the concurrency of data flow notation, new languages would be desirable. Single assignment languages, in which each variable is modified by a single program statement and converted almost directly into data flow form. It might also be useful to develop new algorithms which take advantage of concurrent computation.

As an example, consider the execution of the program fragment
\[ a = (b+c)\cdot(b-c). \]
Figure 3.1 illustrates the sequence of this execution, where a black dot on an arc is to indicate the presence of a data token. The two black dots in Figure 3.1a indicate that the data tokens corresponding to the values of b and c have been generated by some preceding computation. Since b and c are required as inputs for two subsequent nodes two copies of
each data token are generated as shown in Figure 3.1b. Here the availability of the two pairs of inputs enables both the addition and subtraction nodes. Executing in parallel means, each node consumes its input tokens, performs the specified operation and releases the resultant token onto the output arc. Then, the multiplication node is enabled as shown in Figure 3.1c. This node executes and outputs its result corresponding to the identifier a. It is clear that the execution of this program fragment demonstrates the parallel and asynchronous operation of the data flow model.

![Diagram](image)

**FIGURE 3.1:** Execution of a data flow computation \( a = (b+c)(b-c) \)

With the development of microelectronics, the Large-Scale Integration (LSI) technology has increased the number and complexity of components that can fit on a chip. As a result, computers-on-a-chip have emerged; these machines can be used as special purpose devices attached to a conventional computer. The VLSI "systolic" machines discussed in Chapter
represent one class of such machines that have regular structures. Intuitively, a systolic machine is a network of simple and primitive processors that circulate data in a regular fashion. In such machines, each processor pumps data in and out, each time performing some short computations, so that a regular flow of data is kept up in the network. At every processor the control for communication and computation is very simple, and the storage space is only a small constant, independent of the size of the network. For a low-cost and high-performance chip implementation, it is crucial that the geometry of the communication paths in a systolic machine be simple and regular. Generally, systolic machines correspond to synchronous algorithms that use distributed control achieved by simple local control mechanisms and that have small amounts of communications. For the communication overheads and the amount of attention paid to retain as much parallelism as possible, the systolic algorithms, i.e. the algorithms implemented on a systolic array, are considered as most structured amongst the SIMD and MIMD algorithms. While the MIMD algorithms are the least structured. For a systolic algorithm, task modules are simple and interactions among them are frequent. The situation is reversed for MIMD algorithms. Systolic algorithms are designed for direct hardware implementations, whereas MIMD algorithms are designed for executions on general purpose multiprocessors. SIMD algorithms may be considered as lying between these two types of algorithms. The study of how to design algorithms for different parallel machines might reveal that an algorithm requires a peculiar feature of the computer architecture to run at maximum efficiency. Such study may include many questions: How efficient is the algorithm and how much faster is the algorithm than the sequential version or in fact other parallel algorithms?
To answer these questions we need to define some relevant quantities such as the computation time, speed-up and the efficiency of the algorithms.

In real machines, actual computation times are often proportional to the total number of arithmetic operations in the programs. On the other hand, for programs with little arithmetic, the computation time may be proportional to the number of memory accesses or the number of I/O transmissions.

We will use $T_P$ to denote the computation time on a computer with $P$ processors, and $T_1$ to denote the computation time of an uniprocessor (i.e. a sequential) computer.

The term speed-up ($S_P$) of a P processor computer over a sequential computer is defined as:

$$S_P = \frac{T_1}{T_P} \geq 1,$$

and the efficiency ($E_P$) is defined as:

$$E_P = \frac{S_P}{P} \leq 1,$$

where $S_P$ is the maximum possible speed-up using $P$ simultaneous processors. It can be verified that these definitions are consistent with the case when $P=1$ (i.e., the uniprocessor case). In order to achieve a good comparison, the best sequential algorithm for the computation is compared with the best parallel algorithm, even when the two algorithms are quite different.

Stone [1973a] introduced some speed-up ratios and indicated that the best speed-up ratio is linear in $P$, where $P$ is the number of simultaneous processors. This case is applicable to problems that have a natural iterative structure. Such problems include systems of linear equations and other vector and matrix problems. In some cases, problems have speed-up ratios of $O(P/\log P)^*$. These results are less desirable, but such

* $\log P$ is a logarithm to the base 2. From now on, $\log x$ is to mean $\log$ to the base 2 unless otherwise stated.
algorithms are still well-suited for parallel computation. However, algorithms with a speed-up ratio of $O(\log P)$ exhibit very little speed increase when we double the number of processors so that such algorithms are well-suited to serial computers or to computers with less parallelism.

In the literature, we can find many algorithms suitable for SIMD computers. Examples of such algorithms have been introduced by Miranker [1971], Stone [1971],[1973b] and Wyllie [1979]. On the other hand, although implementing algorithms on MIMD computers is more difficult than the implementation of them on SIMD computers as discussed earlier, many parallel algorithms have been developed to run on MIMD computers. For example, Muraoka [1971] showed how parallelism is exploited in applying algebraic expressions on a MIMD computer. Baudet et al [1980], Baudet [1978a] and Dunbar [1978] have developed a variety of parallel algorithms suitable for MIMD computers.

In the rest of this chapter some algorithms for different types of parallel machines will be discussed to exploit the main difference between their organisation, and more emphasis will be paid to MIMD algorithms since the main purpose of this thesis is designing and analysing parallel algorithms for a particular MIMD computer, the NEPTUNE system, for which many algorithms have been considered and developed in the following chapters.
3.2 THE STRUCTURE OF ALGORITHMS FOR PIPELINE AND SIMD COMPUTERS

In this section we describe some algorithmic structures which are suitable for SIMD and pipeline computers. We combine the description of the two types of algorithms here, since the two types of machines implement vector operations as mentioned earlier in Section 3.1. In addition to the surveys of SIMD algorithms, many applications on pipeline computers can be found in the literature.

One of the most successful applications on pipeline computers is the execution of arithmetic operations. Pipeline algorithms for floating-point addition, multiplication, division, and square roots have been discussed in the reference by Chen [1975] and Ramamoorthy and Li [1977]. For these algorithms, the connection among the various stages of the pipeline is linear. For example, the Cray Research CRAY-1 uses six-stage floating-point adders and seven-stage floating-point multipliers and the CDC STAR-100 uses four-stage floating-point adders. For a pipeline floating point adder, the pipe consists of stages for performing exponent alignment, fraction shift, fraction addition, and normalization.

The pipeline approach is ideal for situations where the same sequence of operations will be invoked very frequently, so that the start-up time of the initialisation becomes relatively insignificant. This, however, is the case when the machine is processing long vectors. Actually, one of the main concerns in using pipeline machines such as the CRAY-1 and the STAR-100 is the average length of the vectors to be processed. For integer arithmetic, the bits in the input operands and the carries generated by additions are often pipelined.

Let us consider the following pipeline digit adder using a linear array which is described by Chen [1975]. Suppose that two n-vectors,
Let $U_i = u_{i1}, u_{i2}, \ldots, u_{ik}$ and $V_i = v_{i1}, v_{i2}, \ldots, v_{ik}$ be the binary representations of $U_i$ and $V_i$ respectively. In Figure 3.2, we illustrate how the adder works for $k=3$. The $u_{ij}$ and $v_{ij}$ march towards the processors synchronously as shown.

At each cycle, each processor sums the three numbers arriving from the three input lines and then outputs the carry at the output lines. It is easy to check, from the figure, that when the pair $(u_{ij}, v_{ij})$ reaches a processor, the carry needed to produce the correct $j$th digit in the result of $U_i + V_i$ will also reach the same processor. Therefore, the pipelined adder can compute the sum of $U_i + V_i$ every cycle in the steady state. This algorithm can be applied on a SIMD computer but without considering the binary representations of the digits. The sum of two $n$-vectors described in equation (3.1.1) is then well suited for a SIMD computer. This is because, the sum has $n$ independent computations that can be assigned to $n$ different processors that run simultaneously. Therefore, this algorithm can be computed in one step using $n$ simultaneous processors. This sum can be extended to the addition of two $(n \times m)$ matrices $A$ and $B$, where every row of $A$ is added to every row of $B$ and the required addition is as defined in equation (3.1.2).

If we now consider the matrix multiplication,

$$C = A \cdot B,$$

(3.2.1)

where $A$ is a $(n \times s)$ matrix and defined as:

$$[a_{ij}], i=1,2,\ldots,n \text{ and } j=1,2,\ldots,s,$$

$B$ is an $(s \times m)$ matrix defined as:

$$[b_{ij}], i=1,2,\ldots,s \text{ and } j=1,2,\ldots,m,$$

and the resultant matrix $C$ is defined as:
\[
c_{ij} = \sum_{k=1}^{s} a_{ik} b_{kj}, \quad \text{for } i=1,2,\ldots,n, \quad j=1,2,\ldots,m. \tag{3.2.2}
\]

Therefore, to compute the product, we need to align the columns of B with the rows of A at the processing element of a SIMD computer to obtain the correct results. By doing so, we can compute the inner product as in equation (3.2.2). Obviously, the products A.B and B.A require different types of strategies for accessing the columns and rows of A and B. This problem does not arise on sequential computers, because a sequential computer allows the matrix elements to be fetched in any order. This is one of the problems that arise from parallel computers, particularly, SIMD computers. Therefore, to avoid this inconvenience the array processor (e.g. ILLIAC IV) was proposed. Its processing elements have index registers to provide sufficient flexibility to fetch rows and columns of a matrix with equal ease. This requires a special storing scheme of the matrix known as a skewed scheme (Barnes, et al, 1968).

Consider again the product given in equation (3.2.2). It consists of n.m independent computations to produce the components of the matrix C.

FIGURE 3.2: Pipeline integer adder
Hence, if there are n.m processors, then each component can be assigned to a processor to run simultaneously. However, a pipeline computer such as the CDC STAR-100 does not have index registers in each processor as in the ILLIAC IV. Rather, it is biased to algorithms that process matrices only by rows or only by columns.

For the matrix transpose, the ILLIAC IV cannot transpose the elements easily. Instead, a good interconnection pattern, the "perfect shuffle", can be installed in the ILLIAC computer to facilitate the matrix transpose (Stone, 1971). Therefore, a matrix of size $\sqrt{N} \times \sqrt{N}$ can be transposed by $\log N$ perfect shuffles when $N=2^k$, $k$ any natural number. Meanwhile, the pipeline CDC STAR-100 can compute this problem in $2 \log N$ passes. This is because, it computes the perfect shuffle of a vector in two passes.

It is obvious that, vector and matrix operations are well suited to SIMD computers. This does not stop researchers from implementing other problems in completely different areas. For example, consider the operations of the priority queue. A priority queue is a data structure that can process INSERT, DELETE and EXTRACT-MIN operations and can be used in many programming tasks. If a priority queue is implemented by a balanced tree for example, then an operation of the queue will take $O(\log n)$ time when there are $n$ elements to be stored in the tree (Aho, et al, 1974). This $O(\log n)$ delay can be replaced with a constant delay if a linear array of processors is used to implement the priority queue.

Let the processors of a linear array have the capability of performing comparison-exchange operations on elements in their nearest neighbouring processors. Therefore, the sorting of elements can easily be performed. When an element is inserted into the array from the left it will be transferred to its proper place by trading positions with elements having
smaller values (i.e. the element with the greater value will "bubble along" to the right of the array). The insertion or deletion operations can be carried out concurrently at various processors throughout the array. For example, the second insertion can start right after the first insertion has passed the left-most processor. In this way, any sequence of \( n \) INSERT, DELETE or EXTRACT-MIN operations can be done in \( O(n) \) time on a linear array of \( n \) processors, rather than \( O(n \log n) \) time as required by a uniprocessor.

Some problems that involve "recursive doubling" can successfully be applied on SIMD computers. By recursive doubling we mean that the computation of an algorithm can be broken up into two equal and independent computations of equal complexity, and each capable of simultaneous execution on a SIMD computer. These in turn are reduced to smaller problems recursively. For example, consider the sum of \( n \) numbers which satisfies the following:

Given coefficients \( a_1, a_2, \ldots, a_n \),

let \( x_1 = a_1 \),

then \( x_i = x_{i-1} + a_i \) for \( i \geq 2 \).

The solution for this recurrence problem is given by,

\[ x_n = \sum_{i=1}^{n} a_i. \]

Since the addition operator is associative, we can organise the computation in a manner yielding the smallest parallel running time. If we first assume that \( n \) is a power of 2, we can achieve a time bound of \( \log n \) using \( n/2 \) processors as follows: In the first step of the algorithm, each processor will compute the sum of two elements of the array in unit time. Hence, at least \( n/2 \) processors are required. At the second step, each processor will compute the sum of two sums from step 1 and in this case \( n/4 \)
processors are required. The procedure continues until the final step (i.e. the \((\log n)^{th}\) step) in which only 1 processor computes the complete sum. When \(n\) is not a power of 2, \([n/2]^*\) processors are sufficient to compute the sum in time \([\log n]\). In fact, any associative operation could be used instead of addition, such as multiplication, UNION, MIN, OR operations. Algorithms with the above property can be presented in evaluation trees (Heller, 1978). Figure 3.3 illustrates an evaluation tree of

\[
A = a_1 \circ a_2 \circ \ldots \circ a_n
\]

with the recursive doubling property where \(\circ\) represents any associative operation and \(n\) is given the value 8. At each level of Figure 3.3, the operations are identical and independent, therefore they can be executed simultaneously. As the figure illustrates, \([\log n]\) steps are required to evaluate \(A\) by using \([n/2]\) simultaneous processors. Heller called this algorithm the associative fan-in algorithm and specifically it is known as the log-sum and log-product when the operations are + and \(\times\) respectively.

\[
\begin{align*}
\text{Level 3} &
\begin{array}{c}
\circ \\
\circ \\
\circ \\
\circ
\end{array}
\begin{array}{c}
a_1 \\
a_2 \\
a_3 \\
a_4
\end{array}
\begin{array}{c}
a_5 \\
a_6 \\
a_7 \\
a_8
\end{array}
\end{align*}
\]

\text{FIGURE 3.3: Evaluation tree of Expression A}

* \([x]\) is a ceiling function whose value equals to the greatest integer less than \(x\)*

* \([x]\) equals to the smallest integer greater than \(x\).*
FIGURE 3.4: Evaluation tree of the inner product algorithm

As an example of the associative fan-in algorithm, consider the inner or scalar product of two n-vectors A and B which have the form

\[ S = \sum_{i=1}^{n} a_i b_i , \quad \text{where } A = (a_1, a_2, ..., a_n), \]
\[ \text{and } B = (b_1, b_2, ..., b_n). \]  

(3.2.4)

The evaluation tree for this algorithm is represented in Figure 3.4. In this algorithm, the n products are independent and may be performed simultaneously using n processors. The sum of the resultant products is performed in \( \lceil \log n \rceil \) steps, therefore the inner product algorithm requires \( \lceil \log n \rceil + 1 \) steps using n processors. The extension of this algorithm is the matrix product defined in equation (3.2.2). Since it consists of n.m independent inner products, therefore the matrix C can be evaluated in \( \lceil \log s \rceil + 1 \) steps using n.m.s processors.
For different types of algorithms Wyllie [1979] presented some non-numerical algorithms mainly applied to various data structures. For example, he considered an algorithm that counts the number of elements in a linked list and an algorithm to delete an element from a linked list. In these algorithms, he used a special technique called doubling. Let us consider now the former algorithm in which each processor will be assigned to each element of the list of size L. Each processor will maintain a pointer to an element further along the list, and at each step will "double" the distance its pointer spans by setting its pointer to the value of the pointer belonging to the element pointed to on the previous iteration.

To analyse the running time of this algorithm, notice that the span for each processor is computed as the sum of two spans from the previous iteration, except possibly for the last, when the null link at the end of the list is discovered. Therefore, the running time is $O(\log L)$.

Another widely used non-numerical problem is the sorting of a number of keys. The most well-known algorithm which uses a linear processor array is the odd-even transposition sort (Knuth, 1973). The algorithm commences by storing the n keys to be sorted in ascending order in a linear processor array. The problem can be solved in n steps. The odd- and even-numbered processors are activated alternately. Assume that the even numbered processors are activated first. In each cycle, the comparison exchange operations are performed as follows: the key in every activated processor is compared with the key in its right-hand neighbouring processor, and then the key with the smaller value is stored in the activated processor. Hence, within n cycles, the keys will be sorted in the array as shown in Figure 3.5.

From all the above-mentioned algorithms, we conclude that in developing a parallel algorithm it is assumed that the computer has unlimited parallelism.
This leads to an algorithm that requires an unrealistically large number of processors. When no such large number of processors are available on a machine, a more practical algorithm should be constructed in which the processor requirement is reduced to a realistic number. A well constructed algorithm is one that does not reduce the efficiency of the algorithm with a large number of processors. For this idea, we explain two basic principles; the algorithm decomposition and the problem decomposition (Hyafil and Kung, 1974). In the algorithm decomposition principle, if the original algorithm performs $q_i$ operations during step $i$, then the converted algorithm will require $\left\lceil \frac{q_i}{P} \right\rceil$ steps to perform step $i$ of the original algorithm, where $P$ is the maximum number of available processors. This means, the decomposition takes place in each step. On the other hand, in the problem decomposition the original problem of order $n$ is partitioned into smaller problems of order $P$ and then the parallel algorithm is applied to each of the smaller problems.
To illustrate these principles, consider first the odd-even transposition sort discussed earlier. It can be generalised so that each processor holds a sorted subsequence of keys rather than a single key (Baudet and Stevenson, 1978). This generalisation will be extensively discussed in Chapter 6. In this case, the algorithm uses the merge-splitting operation instead of the comparison-exchange operation. However, the required run-time to sort \( n \) keys on \( k \) linearly connected processors is given by,

\[
O((n/k \log(n/k)) + O(k(n/k)),
\]

provided that each processor can hold \( n/k \) keys. In the formula (3.2.5), the first term is the time required to sort \( n/k \) subsequence in each processor, and the second term is for the time to perform the odd-even transposition sort on \( k \) sorted \( n/k \) subsequences. Clearly, when \( n \) is large relative to \( k \), a linear speed-up (i.e. a speed-up ratio near \( k \)) is obtained. This optimal speed-up is due to the fact that when \( n \) is large relative to \( k \), the computation performed within each processor is large compared to the inter-processor communication. Therefore, the overheads arising from interprocessor communication become relatively insignificant.

Now, consider again the sum of \( n \) numbers represented in the evaluation tree of Figure 3.3, which can be viewed in another way. Suppose that \( P \leq \frac{n}{2} \) processors are available. Then, to compute the sums of level 1 would require at most \( \left\lfloor \frac{n}{2p} \right\rfloor \) time, the next higher level would require at most \( \left\lfloor \frac{n}{4p} \right\rfloor \) time, etc. until the final level requires \( \left\lfloor \frac{n}{2 \log n_p} \right\rfloor \) time.

Therefore, the total required run-time is,

\[
T \leq \left\lfloor \frac{n}{2p} \right\rfloor + \left\lfloor \frac{n}{4p} \right\rfloor + \ldots + \left\lfloor \frac{n}{2 \log n_p} \right\rfloor 
\leq (1 + \frac{n}{2p}) + (1 + \frac{n}{4p}) + \ldots + (1 + \frac{n}{2 \log n_p}) 
= \left\lfloor \log n \right\rfloor + \frac{n}{p} \left( 1 + \frac{1}{2 \log n} \right) 
\leq c_1 \log n + c_2 \cdot \frac{n}{p} 
= O(\log n + \frac{n}{p}) \text{ units.}
\]

(3.2.6)
Obviously, T is $O(\log n)$ when $P = \frac{n}{\log n}$, so a running time of the same order as the previous algorithm can be achieved with fewer processors reduced by a logarithmic factor.

If we apply the same concept to the algorithm of counting the number of elements in a linked list which is mentioned above, we see that, unlike the sum algorithm, no gain in the efficiency of the algorithm is obtained when fewer processors are cooperating. To clarify this further, suppose that $P < L + 1$ processors are available and assigned to the list elements such that each list element does the span once before any element is allowed to do that again. This modification is correct, but it requires $\left[ \frac{L}{P} \right]$ time to ensure that each processor at least doubles its span. Therefore, the total running time becomes of $O(\frac{L}{P} \log L)$. Since $P$ is included multiplicatively rather than additively in this formula, then the run-time cannot be better than the original version of this algorithm.

In conclusion, numerous algorithms have been developed for SIMD computers using the two basic techniques explained above, most of which are implemented on a two-dimensional array of processors. A typical problem that has been investigated is the solution of a linear system of equations (Pease, 1967 and Csanky, 1975). Also, particular forms of linear systems have also been investigated such as tridiagonal systems (Stone, 1973a, 1973b and Heller et al, 1974). Systems of equations arising from differential equations have been studied by many authors, e.g. Hayes [1974]. However, relaxation methods for solving partial differential equations (see Chapter 4) match the square array structure evenly. Typically, the variable $u_{ij}$ representing the solution at the mesh point $(i,j)$ is updated by a differential equation of the form,

$$u_{ij} = F(u_{i+1,j}, u_{i,j-1}, u_{i,j}, u_{i,j+1}, u_{i-1,j}, u_{i-1,j-1}).$$
Hence, if \( u_{ij} \) is stored at processor \((i,j)\) of the processor array, then each update (or iteration) involves communications only among its neighbouring processors. The central control provided by SIMD machines such as the ILLIAC IV is useful for broadcasting the relaxation and termination parameters, which are often needed in these relaxation methods.

Parallel forms of Fast-Fourier transforms have also been developed (Stone, 1971). Some other various parallel algorithms have been investigated such as cellular algorithms for pattern recognition (Smith, 1971), algorithms for graph problems (Levitt and Kautz, 1972), sorting algorithms studied by Thompson and Kung [1977], and algorithms for eigenvalue problems (Barlow, et al, 1982b).

All these algorithms explore the potentiality of SIMD and pipeline computers although such computers have a large amount of synchronisation and inter-processor communication which are of smaller effect on MIMD computers.
3.3 **THE STRUCTURE OF ALGORITHMS FOR MIMD COMPUTERS**

In this section, some parallel algorithms suitable for MIMD computers are discussed. Some of them are easy to decompose into an independent number of processes to be carried out by several processors with a minimum of interference amongst them. While some other algorithms require special case since they are decomposed into cooperating processes which are executed simultaneously. The later algorithms are called asynchronous which is the characteristic of algorithms for asynchronous MIMD computers. For the former type of algorithms, we consider expressions of the form of equation (3.2.3), that is,

\[ A = a_1 \circ a_2 \circ \cdots \circ a_n , \]

which is evaluated using the associative fan-in algorithm used in SIMD computers. From Figure 3.3 above, it is obvious that the operations at each level are independent, but the operations between the levels are not. Therefore, from the asynchrony property of the MIMD computers, this algorithm can be restructured in a way so as to remove as much of the dependency as possible. Since, on designing an algorithm, it is assumed that the multiprocessor systems have \( P \) processors, therefore the above expression can be partitioned into \( P \) subsets. Thus,

\[ A = (a_1 \circ a_2 \circ \cdots \circ a_{\ell}) \circ (a_{\ell+1} \circ \cdots \circ a_{2\ell}) \circ \cdots \circ (a_{(P-1)\ell} \circ \cdots \circ a_{n}) , \]  

where \( \ell = \left\lfloor \frac{n}{P} \right\rfloor \). The evaluation tree of this algorithm is presented in Figure 3.6. By using the $DOPAR$/PAREND construct, this algorithm can be programmed as in program 3.1 below.
FIGURE 3.6: Evaluation tree of Expression A in equation (3.3.1)

Program 3.1

\begin{verbatim}
NEIM=N/P
M=log p
$DOPAR 10 IP=1,P
   X(IP)=0
   I=NEIM*(IP-1)+1
   J=NEIM*IP
   DO 20 K=I,J
      X(IP)=X(IP)*A(K)
   20 CONTINUE

10 $PAREND
   DO 30 NP=1,M
      L=P/2**NP
   $DOPAR 40 K=1,L
      IL=2*K-1
      I2=2*K
      X(K)=X(IL)*X(I2)
   40 $PAREND
   CONTINUE
\end{verbatim}

Obviously, in this program, the P paths will run concurrently. At the end of each path I, where I=1,2,...,P, X(I) contains the partial result of the computation of the subexpression which corresponds to that which is evaluated by path I. When all paths have been completed the total value of the whole expression is evaluated from all the X(I)'s evaluated in parallel by log p steps.

This algorithm is analysed as follows:
In order to find the speed-up and efficiency ratios it is required to find the sequential and the parallel running-time $T_1$ and $T_p$ respectively. Thus,

$$T_1 = n-1 \text{ operations,}$$

and the complexity measured for each processor requires at most:

$$T_p = \left\lfloor \frac{n}{p} \right\rfloor -1 + \log P \text{ operations.}$$

The final step of the algorithm where the $P$ values of $X$ are computed requires $\log p$ operations. Hence, the term 'log $p$' appears in the $T_p$ formula. The speed-up and the efficiency then become,

$$S_p = \frac{T_1}{T_p} = \frac{n-1}{\left\lfloor \frac{n}{p} \right\rfloor -1 + \log p}$$

$$< \frac{n-1}{\frac{n}{p} + \log p} , \text{ since } \left\lfloor \frac{n}{p} \right\rfloor > \frac{n}{p} - 1$$

$$< P - \frac{P \log p - 2p + 1}{\frac{n}{p} + \log p} = P[1 - \frac{P \log p - 2p + 1}{n + p \log p - 2p}] , \quad (3.3.2)$$

and

$$E_p < 1 - \frac{P \log p - 2p + 1}{n + p \log p - 2p} . \quad (3.3.3)$$

It can be guaranteed that the fraction $\frac{P \log p - 2p + 1}{n + p \log p - 2p} < 1$ if $n \gg p$. Thus, the speed-up is nearly $P$ (i.e. linear in $P$), and $E_p$ is approximately 1.

From equation (3.3.2), it can be clarified that for a fixed value of $n$, the speed-up as well as the efficiency slightly decreases as $P$ increases. This is due to process allocation overheads, and some other overheads which will be clarified in the following chapters.

The same strategy of decomposition can now be applied to the scalar product of two $n$-vectors of equation (3.2.4). Supposing, there are $P$ processors, then the $P$ subsets of size $i = \left\lfloor \frac{n}{P} \right\rfloor$ have independent computations and can be evaluated simultaneously. This algorithm proceeds by evaluating the $i$ multiplication operations of each subset concurrently, then it
performs the addition operations in \( \log p \) steps. The program is given in Program 3.2 below:

Program 3.2

```
L=N/P
M=\log P
$DOPAR 10\ IP=1,P
\ Il=L*(IP-1)+1
\ I2=L*IP
IF(IP.EQ.P)\ I2=N
\ C(I)=0
DO 20\ J=Il,I2
\ C(J)=C(Il)+A(J)*B(J)
\ C(Il)=C(J)
20\ CONTINUE
$PAREND
DO 40\ NK=1,M
\ NL=P/2**NK
$DOPAR 50\ K=1,NL
\ J1=2*K-1
\ J2=2*K
\ C(K)=C(J1)+C(J2)
50\ $PAREND
40\ CONTINUE
```

The complexity of this algorithm is given by,

\[ T_1 = n(M + A) \text{ operations,} \]

and

\[ T_p = \left\lfloor \frac{n}{p} \right\rfloor (M + A) + A \log P, \]

where \( M \) and \( A \) are the times required to perform a multiplication and an addition operation respectively. This leads to the following:

\[
S_p = \frac{n(M + A)}{n(M+A)+A \log p} < \frac{n(M + A)}{n(M+A)-1+A \log p}, \text{ since } \left\lfloor \frac{n}{p} \right\rfloor > \frac{n}{p} - 1
\]

\[
< \frac{p}{p} - \frac{p(A \log p - p)}{n(M+A)-p+A \log p}
\]

\[
< p(1 - \frac{A \log p - p}{n(M+A)-p+A \log p})
\]

The efficiency then becomes:

\[
E_p < 1 - \frac{A \log p - p}{n(M+A)-p+A \log p}
\]
where both the efficiency and the speed-up satisfy the given rules in the formulae (3.1.3) and (3.1.4).

Now, let us consider the matrix operations, addition, subtraction and multiplication. First, we consider the addition of two \((n \times m)\) matrices such that,

\[
C = A + B,
\]

which is defined as,

\[
c_{ij} = a_{ij} + b_{ij} \quad \text{for } i=1,2,\ldots,n,
\]

\[
j=1,2,\ldots,m.
\]

The computations of the elements \(c_{ij}\)'s are independent, therefore they can be implemented in many ways to assign these computations to different processors. When only two processors are cooperating, we can assign the odd numbered rows (columns) of \(C\) to one processor and the even numbered ones in the other processor. However, when there are \(p\) processors available, \(\left\lfloor \frac{n}{p} \right\rfloor\) rows (columns) are assigned to each processor which in turn carries out the evaluation sequentially.

It is easy to see that the required run-time when \(P\) processors are used is given by,:\n
\[
T_p = \left\lfloor \frac{n}{p} \right\rfloor \cdot m \text{ operations},
\]

and

\[
T_1 = n \cdot m \text{ operations}.
\]

Then,

\[
S_p = \frac{n \cdot m}{\left\lfloor \frac{n}{p} \right\rfloor \cdot m} = \frac{nm}{\frac{n}{p} + 1}, \quad \text{since } \left\lfloor \frac{n}{p} \right\rfloor < \frac{n}{p} + 1
\]

\[
= p \left[ 1 - \frac{p}{nm+p} \right],
\]

and

\[
E_p = 1 - \frac{p}{n \cdot m + p}.
\]

When \(n \cdot m >> p\), \(S_p\) becomes less than \(p\) and \(E < 1\). In fact, \(S_p\) is not exactly equal to \(P\) in all the algorithms discussed above and this is due to the overheads incurred by the path allocation and interprocessor communication.
The matrix multiplication of equation (3.2.2), where each component of the resultant matrix is a scalar product, can be programmed using the same strategy discussed above. By using the scalar product algorithm presented in program 3.2, each component of the resultant matrix at a time is then evaluated.

From the algorithms discussed earlier, we notice the difference in the implementation on SIMD and MIMD computers. This follows from the difference in the characteristic of the two types of computers. It is known that the processors of a SIMD computer are synchronised as well as being synchronous, i.e. each processor executes the same instructions, and the instructions are all executed at exactly the same time. In MIMD computers, the processors are not exactly identical and often they differ in their speeds. Therefore, even if the instruction streams are identical, the processors may not execute each instruction at exactly the same time. In fact, in the design and analysis of parallel algorithms for asynchronous multiprocessors, one should assume that the time required to execute the steps of a process carried out by one processor is unpredictable (Kung, 1976). However, there are some major sources for causing fluctuations in the execution times of the processes. These include variations in the computation time due to different instances of inputs, memory conflicts and variations in the individual processor speeds. In addition, other factors such as the operating system's scheduling policies that assign certain processors to perform I/O terms, allocate processors to processes, switch a processor from one process to another, and so on need to be taken into consideration. The asynchronous behaviour leads to serious issues regarding the correctness and efficiency of an algorithm. The correctness issue arises because during the execution of an algorithm operations from
different processes may interleave in an unpredictable manner. The efficiency issue arises because any synchronisation introduced for correctness reasons takes extra time and also reduces concurrency.

On a MIMD computer, synchronous and asynchronous algorithms may be performed. The synchronous algorithms are algorithms with processes that contain so called "interaction points" by which the process can communicate with other processes. These interaction points divide a process into stages. Therefore, at the end of each stage a process may communicate with other processes before starting the next stage. Hence the need for synchronisation at these interaction points. This actually degrades the performance of the algorithm, and hence needs to be considered in the development of asynchronous algorithms. An asynchronous algorithm is a parallel algorithm where communications between processes are achieved through the global variables or shared data. In many cases, to ensure logical correctness, the operations on the shared data are programmed as critical sections. The main characteristic of an asynchronous algorithm is that its processes never wait for inputs at any time but continue or terminate according to whatever information is currently contained in the shared data. However, processes may be blocked from entering critical sections since the access to critical sections follows the First-In-First-Out (FIFO) rule as was discussed in Chapter 2.

Now, we consider some asynchronous algorithms. An asynchronous zero-searching algorithm with two processes for finding the root of a continuous function $f$ has been developed by Kung [1976]. This algorithm is based on the Fibonacci rule and is described as follows:

Let $\theta$ be the positive root of the equation $\theta^2 + \theta = 1$, i.e. $\theta = 0.168$ is the reciprocal of the golden ratio $\phi$, and suppose that the interval of
uncertainty has length $\ell$. There are two types of states, which are illustrated in Figure 3.7, called $A_1(\ell)$ and $A_2(\ell)$ respectively. $A_1(\ell)$ indicates that one of the two processes is evaluating the function at the point "0" which is $\theta^2 \ell$ from one end of the interval $\ell$. $A_2(\ell)$ indicates that both processes are evaluating the function at the point "0" each $\theta \ell$ from either end of the interval $\ell$. Suppose that the algorithm commences at the state $A_2(\ell)$ and the evaluation at the left point will finish first. Then after the completion of the first point's evaluation, the new interval of uncertainty is either $[0-\infty)$ or $(-\infty),_0$, depending upon the output. If the first case occurs, then the process which has just completed the evaluation at the left point activates a new evaluation at the point "A", which is defined by the following graph:

![Graph 1](image1)

Hence the state $A_1(\theta^2 \ell)$ is obtained. Similarly, state $A_2(\theta \ell)$ can arise from the second case as illustrated in the following graph:

![Graph 2](image2)

Hence the state $A_2(\theta \ell)$ is transited to either state $A_1(\theta^2 \ell)$ or $A_2(\theta \ell)$. This is denoted by,

$$A_2(\ell) \rightarrow A_1(\theta^2 \ell) \lor A_2(\theta \ell). \quad (3.3.4)$$

Similarly, the state $A_1(\ell)$ can be transited as follows:

$$A_1(\ell) \rightarrow A_1(\theta^2 \ell) \lor A_1(\theta \ell) \lor A_2(\ell). \quad (3.3.5)$$

The state $A_2(\ell)$ of the transition rule (3.3.5) corresponds to the assignment of the second process at the indicated point "A" of the graph below:
The other two states in formula (3.3.5) corresponds to the termination of the evaluation of the first process before this assignment can be made. Suppose that this algorithm starts from state $A_1(\ell)$, then its associative transition tree is presented in Figure 3.8.

![Diagram](image)

**FIGURE 3.7:** The two states of the zero-searching algorithm

![Diagram](image)

**FIGURE 3.8:** A transition tree for the asynchronous zero-search algorithm

The algorithm passes through all the states along one of the paths in the tree. This means, the search consists of the usual sequence of function evaluations and concomitant reductions of the interval of uncertainty. The asynchrony being the specification of each reduction upon the completion of an evaluation of one process while ignoring and usually interrupting the other process. The recurrent nature of states as presented in the transition
rules (3.3.4) and (3.3.5) shows the coherency of the asynchronous search process. The asynchronous zero-search algorithm can be analysed easily since it is associated with a transition tree of Figure 3.8.

If \( N \) is the number of function evaluations completed by the algorithm, and since the evaluations are performed by two concurrent processes, then the expected time by the algorithm is \( \frac{N \bar{t}}{2} \), where \( \bar{t} \) is the mean time for the sub-intervals. Therefore, the speed-up ratio between the expected time taken by the binary search algorithm and the asynchronous zero-search algorithm is given by:

\[
S_P = \frac{\bar{t} \log \frac{L}{N}}{2 \log \frac{L}{N}}, \quad \text{as } N \to \infty.
\]

The value of \( N \) can be determined for which the efficiency of the algorithm depends. Clearly, the value of \( N \) in the worst case is given by the length of the longest path in the transition tree, in the best case by the length of a shortest path, and in the average case by the average path length. Therefore, in the worst case \( N = \log \frac{L}{\bar{t}} \), and \( S_P = -1.388 \). Thus, this algorithm is superior to a two process synchronised search algorithm if the average waiting time in the synchronised case exceeds approximately 14% of the evaluation time. Similar results for the best and average cases are given by (Kung, 1976).

A generalisation to this algorithm has been studied by Kung [1976], where the algorithm is generalized to be suitable for 3 or more processes. In fact, a four process implementation has been studied by Staunstrup [1981] where successful results were obtained. In the case of 3 processes, the following two patterns are considered:

\[
\begin{align*}
\frac{L}{4} & \quad \frac{L}{4} & \quad \frac{L}{4} & \quad \frac{L}{4} \\
\frac{L}{3} & \quad \frac{L}{6} & \quad \frac{L}{6} & \quad \frac{L}{3}
\end{align*}
\]

FIGURE 3.9: Two patterns for the algorithm involving 3 processes
Generally speaking, an asynchronous zero-search algorithm with $k$ processes corresponds to an asynchronous algorithm with $(k-1)$ processes for locating the maximum of an unimodal function. Thus, the patterns of Figure 3.9 give an asynchronous algorithm with two processes for locating the maximum of an unimodal function which is faster than the optimal synchronised algorithm with two processes (Karp and Miranker, 1968) as long as the penalty factor of synchronisation is greater than one.

Another example for MIMD computers is the chaotic relaxation method for solving a system of equations in an asynchronous manner (Chazan and Miranker, 1969). If $A$ is symmetric and positive definite ($m \times m$) matrix, then solving the linear system,

$$Ax = d$$

by relaxation may be described as successive univariate minimization, usually in coordinate directions of the quadratic form,

$$Q(x) = \frac{1}{2}(x,Ax) - (d,x).$$

For the Gauss-Seidel method, each univariate minimization is completed prior to starting any other (see Chapter 4). Suppose now, that there are $r$ processors available for this computation where each processor may be assigned to effect a minimization. However, during the time that a processor is determining what displacement at a point $x$ in the space $\mathbb{R}^n$ is to be effected, other processors may complete their tasks and by writing into the memory, displace the point $x$. Thus the displacement to a minimum determined by a processor at the point $x$ may be made at some other point. In general, the processors will rapidly get out of phase and the whole process will take on a chaotic appearance. Rosenfeld and Driscoll [1969] applied numerical experiments based on solving the Dirichlet problem (to be described in Chapter 4) using the above process. Also, Donnelly [1971] extended some of this work for the special case termed periodic chaotic relaxation.
Other numerical applications are the asynchronous iterative methods developed by Baudet [1978b]. These algorithms turned out to be good algorithms compared with the synchronised algorithms. More details of such algorithms are presented in Chapter 4. Baudet [1980] carried out his research on the development of asynchronous algorithms and introduced an algorithm to execute a sequence of linearly ordered tasks on an asynchronous multiprocessor in which two cases were considered: the implementation with and without critical sections. By taking advantage of the fluctuations in the execution times of individual tasks, his algorithm achieved a substantial speed-up when \( k \) processes were cooperating. To execute \( w_1, \ldots, w_n \) tasks by \( P \) processes the following algorithm was implemented.

Whenever a process \( P \) is ready to execute a task, then,

(a) if no task has yet been completed by any process, process \( P \) starts executing task \( w_1 \),

(b) otherwise, if the last task \( w_n \) has not yet been completed by any process, process \( P \) starts executing a task which is unfinished and ready for execution.

Let \( t_i \) for \( i=1,2,\ldots \) with \( t_1 < t_{i+1} \) be the time instants of completions of the tasks by the processes. Figure 3.10 illustrates a possible scheduling of the tasks when they are executed by three processes.

![Figure 3.10: A possible task scheduling with three processes](image-url)
We notice that, when process \( P_3 \) finishes task \( w_3 \) at time \( t_8 \), process \( P_2 \) has already completed \( w_4 \). Thus, after \( P_3 \) completes \( w_3 \), it starts executing \( w_5 \) rather than \( w_4 \). Thus, the algorithm achieves a speed-up not by sharing work amongst processes, but by taking advantage of fluctuations in the execution times. The analysis of this algorithm depends on some probability concepts which are also used in our analysis of the algorithms presented in this thesis as we will show in the following chapters.

Since the \( t_i \)'s are random variables, then we define \( \tau_p(i) \) to be the mean time required for the first completion of the task \( w_i \). We define, \( T_1(n) \) as the sequential run-time for a sequence of \( n \) tasks and \( T_p(n) \) as the run-time with \( P \) processes.

Therefore,

\[
T_1(n) = \tau_1(1) + \tau_1(2) + \ldots + \tau_1(n).
\]

Since the \( \tau_1(i) \) are independent and identically distributed with mean \( \tau \), we deduce,

\[
\overline{T}_1(n) = n\tau . \tag{3.3.6}
\]

Now, to evaluate \( \overline{T}_p(n) \), we assume that the random variable \( T \) is exponentially distributed with mean \( \tau \). This is because the solution requires finding a Markov chain in the evaluation of the processes which become possible if the probability distribution for the execution times of the tasks is exponential. Also, it was found that with the assumption of our exponential distribution a better speed-up was obtained compared with that of the uniform or Dirac distributions (Baudet et al, 1980).

Further to the exponential assumption, we assume that all processes start at the same time \( t=0 \) (i.e. all processes start the execution of task \( w_1 \)).

Thus,

\[
\overline{T}_p(n) = \overline{T}_p(1) + \overline{T}_p(2) + \ldots + \overline{T}_p(n) . \tag{3.3.7}
\]

Since all processes start execution at task \( w_1 \), then \( \overline{T}_p(1) \) is given by
the minimum of $P$ random variables distributed as $T$, and since $T$ is exponentially distributed the minimum has the mean:

$$
\bar{T}_P(1) = \frac{1}{P}.
$$

For $i=1,2,\ldots,n-1$, the mean of $\bar{T}_P(i+1)$ is given by,

$$
\bar{T}_P(i+1) = \frac{P}{P} \sum_{j=1}^{P} q_{P,j} \left( \frac{1}{P} \right) = \frac{1}{P} \sum_{j=1}^{P} j q_{P,j},
$$

where $q_{P,j}$ is the probability that the first completion of task $w_{i+1}$ is equal to $t_{i+1}$ for some $i$.

From equations (3.3.7) and (3.3.8) we obtain:

$$
\bar{T}_P(n) = \frac{1}{P} \left( 1 + (n-1) \sum_{j=1}^{P} j q_{P,j} \right).
$$

Let $N_P = \sum_{j=1}^{P} j q_{P,j}$, then the speed-up ratio becomes,

$$
S_P(n) = \frac{\bar{T}_P(n)}{\bar{T}_P(n)} = \frac{nr}{1 + (n-1)N_P}.
$$

When both $n$ and $p$ are large, Baudet has shown that $S_P(n) \approx 0.7981P$.

This is achieved when no critical sections are implemented. A slightly different analysis for the case of critical sections is also presented in the above reference. Baudet's analytic results of the algorithm with critical sections demonstrate that the parallel algorithm using two processes is not necessarily faster than the sequential algorithm, because of the critical section overheads. This confirms that, in practice, the speed-up ratio does not necessarily increase as the number of processes increases.

Asynchronous multiprocessors can support truly concurrent data base systems, where simultaneous access to a data base by more than one process is possible (Kung, 1980). In a concurrent data base system, a number of
on-line transactions are allowed to run concurrently on a shared data base. One of the important issues arising from the concurrent execution of transactions is the consistency problem. A data base is said to be consistent if all integrity constraints defined for the data are met. Two approaches may be implemented. One approach requires no knowledge of the integrity constraints, but requires syntactic information about the transactions. The second approach requires a knowledge of the integrity constraints to construct correct and more efficient concurrent data base systems. In Kung and Papadimitriou [1979] maximum degrees of concurrency are proved to depend upon the types of knowledge that are available. These two approaches are discussed by Kung [1980]. Also, Kung described algorithms based on search trees which exploit high asynchrony. For example, algorithms for a binary search tree that can support concurrent searching, insertion and deletion of items on the tree are proposed. In such algorithms, any process should lock a small constant number of nodes at a given time and writer-exclusion locks are used simply to prevent the obvious problems created by simultaneous updates of a node by more than one process. The algorithm achieves a high degree of concurrency in the following sense.

An update just does whatever an insertion or deletion must do, and postpones the work of rebalancing the unbalanced structure caused by the updating. Other processes can perform the postponed work on separate processors. By this postponement, the multiprocessing capability of a multiprocessor environment can be utilized. The same idea is used in garbage collection. Rather than performing the garbage collection itself, the deleter appends all deleted nodes to a list of nodes to be garbage collected later. In this way, the deleter need not wait until it is safe to do the garbage collection (i.e., the time when no one else will access the deleted nodes), and the garbage collection can be done by separate
processors. Another important point raised in these algorithms is that a process makes updates only on a local copy of the relevant portion of the tree and later introduces its copy into the global tree in one step. With this technique one can get the effect of making many changes to the data base in one indivisible step without having to lock a large portion of the data. However, this creates the problem of backing up processors that have read data from old copies. It turns out that because of the particular property of the tree structure, the back up problem can be handled efficiently.

One problem raised from multiprocessor systems is that the efficiency analysis of algorithms for such computers is usually difficult, since execution times are random variables rather than constants. Typically, techniques in order statistics and queueing models must be employed (e.g. Robinson, 1979). Such techniques are used in our analysis of the asynchronous iterative methods described in Chapter 4. Generally speaking, algorithms with large module granularity are well suited to asynchronous multiprocessors. In this case, a process can proceed for a long period of time before it must wait for input from other processes.
3.4 THE STRUCTURE OF ALGORITHMS FOR VLSI SYSTEMS

As a result of the new technology that has developed Very Large Scale Integrated (VLSI) circuits on 'chips' which was described in Chapter 1, it became feasible to build low-cost, special-purpose peripheral devices to rapidly solve sophisticated problems. In general, VLSI offers excellent opportunities for inexpensive implementation of high-performance devices. Thus, the performance in VLSI technology not only refers to the traditional operation counts as in the classical analysis of algorithms, but the throughput obtainable when a special-purpose peripheral device is attached to a general-purpose host computer. This means that time spent in I/O, control, and data movements as well as the arithmetic must all be considered.

It is clear that any simple interconnection pattern results in a successful implementation with high achievements. Therefore, parallel algorithms that have simple and regular data flows have been implemented on VLSI processor arrays that are capable of pipelining matrix computations with optimal speed-up (Mead and Conway, 1980). These processor arrays have size depending only on the bandwidth of the matrix to be processed and is independent of the length of the band. Therefore, a fixed-size processor array can pipeline band matrices with arbitrarily long bands.

One of the array connections used by Mead and Conway is the hexagonally mesh-connected (or hex-connected) scheme (Figure 3.11). This scheme is natural for matrix problems.

Consider, for example, the matrix multiplication of two \((n \times n)\) matrices as in equation (3.2.1). The matrix multiplication can be computed by the following recurrence:

For all \(1 \leq i, j \leq n\)

\[
\begin{align*}
\mathbf{c}^{(1)}_{ij} &= 0 \\
\mathbf{c}^{(k+1)}_{ij} &= \mathbf{c}^{(k)}_{ij} + a_{ik} b_{kj}, \quad k=1,2,...,n \\
\mathbf{c}^{(n+1)}_{ij} &= \mathbf{c}^{(n)}_{ij}
\end{align*}
\]
Let A and B be \((n \times n)\) matrices of band width \(w_1\) and \(w_2\), respectively (see Chapter 4 for the bandwidth definition). The algorithm is to evaluate the recurrence above by pipelining the \(a_{ij}\), \(b_{ij}\) and \(c_{ij}\) through an array of \(w_1w_2\) hex-connected processors. A matrix multiplication scheme of Figure 3.12 is to be evaluated on a diamond-shaped interconnection network which is represented in Figure 3.13. In this Figure, the processors are hex-connected and data flows are indicated by arrows.

\[
\begin{bmatrix}
  a_{11} & a_{12} & \cdots & a_{1w_1} \\
  a_{21} & a_{22} & \cdots & a_{2w_1} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{w_11} & a_{w_12} & \cdots & a_{w_1w_1} \\
\end{bmatrix}
\begin{bmatrix}
  b_{11} & b_{12} & \cdots & b_{1w_2} \\
  b_{21} & b_{22} & \cdots & b_{2w_2} \\
  \vdots & \vdots & \ddots & \vdots \\
  b_{w_21} & b_{w_22} & \cdots & b_{w_2w_2} \\
\end{bmatrix}
\begin{bmatrix}
  c_{11} & c_{12} & \cdots & c_{1w_2} \\
  c_{21} & c_{22} & \cdots & c_{2w_2} \\
  \vdots & \vdots & \ddots & \vdots \\
  c_{w_21} & c_{w_22} & \cdots & c_{w_2w_2} \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
  c_{11} & c_{12} & \cdots & c_{1w_2} \\
  c_{21} & c_{22} & \cdots & c_{2w_2} \\
  \vdots & \vdots & \ddots & \vdots \\
  c_{w_21} & c_{w_22} & \cdots & c_{w_2w_2} \\
\end{bmatrix}
\]

**FIGURE 3.11:** A hexagonal array

**FIGURE 3.12:** Band matrix multiplication

The elements in the bands of A, B and C move through the network in three directions synchronously. Each \(c_{ij}\) is initialised to zero as it enters the network through the bottom boundaries. (For the general problem of computing \(C = AB + D\), where \(D\) is a \((n \times n)\) matrix, \(c_{ij}\) should be initialised as \(d_{ij}\)). One can easily see that with the inner product step processors depicted in Figure 3.14, each \(c_{ij}\) is able to accumulate all its terms.
FIGURE 3.13: The hexagonal array for the matrix multiplication problem shown in Figure 3.12
before it leaves the network through the upper boundaries. The inner product step processor has three registers $R_A$, $R_B$ and $R_C$, and has six external connections, three for input and three for output. In each unit time interval, the processor shifts the data on its input lines denoted by $A$, $B$ and $C$ into $R_A$, $R_B$ and $R_C$ respectively, computes $R_C + R_A \cdot R_B'$ and makes the input values for $R_A$ and $R_B$ together with the new value of $R_C$ available as outputs on the output lines denoted by $A$, $B$ and $C$, respectively. All outputs are latched and the logic is clocked so that when one processor is connected to another, the changing output of one during a unit time interval will not interfere with the input to another during this time interval.

Now, by referring back to the matrix multiplication algorithm, an array of $w_1 \cdot w_2$ hex-connected processors with the above characteristics can pipeline the matrix multiplication $A \cdot B$, in $3n + \min(w_1, w_2)$ units of time, where $A$ and $B$ are of size $(n \times n)$ and they are of bandwidth $w_1$ and $w_2$ respectively. Note that in any row or column of the network, out of every three consecutive processors, only one is active at any given time. It is possible to use about one third of the $w_1 \cdot w_2$ processors in the network for multiplying two matrices with bandwidth $w_1$ and $w_2$.

Now, if $A$ and $B$ are $(n \times n)$ dense matrices, (see Chapter 4 for the definition) then $3n^2 - 3n + 1$ hex-connected processors can compute $A \cdot B$ in $5(n-1)$
units of time. In this case, the bandwidth of such matrices is \( w = 2n - 1 \).

If this \( w \) is so large that a corresponding algorithm requires more processors than a given network provides, then one should decompose the matrix and solve each subproblem on the network.

An important application of the above matrix computation is that an \( n^2 \)-point Discrete Fourier Transform (DFT) can be computed by first performing \( n \) independent \( n \)-point DFTs and then using the results to perform another set of \( n \) independent \( n \)-point DFTs. The computation of any of these two sets of \( n \) independent \( n \)-point DFTs is simply a matrix multiplication \( A \cdot B \) where the \((i,j)\) entry of matrix \( A \) is \( w^{(i-1)(j-1)} \) and \( w \) is a primitive \( n \)th root of unity. Thus, when \( O(n^2) \) hex-connected processors are used, an \( n^2 \)-point DFT can be computed in \( O(n) \) time.

On the other hand, one can often reduce the number of processors required by an algorithm if the matrix is known to be sparse or symmetric. The matrices arising from a set of finite difference or finite element approximations to differential equations are usually "sparse band matrices". These are band matrices whose non-zero entries appear only in a few of those lines in the band which are parallel to the diagonal. In this case, by introducing appropriate delays to each processor for shifting its data to its neighbours, the number of processors required by the algorithm above can be reduced to the number of those diagonal lines that contain non-zero entries. This variant is, of course, useful for performing iterative methods involving sparse band matrices.

Other algorithms involving matrix operations are implemented on a hex-connected array, such as the LU-decomposition of a matrix and solving triangular linear systems of equations. On the other hand, problems that can be formulated as matrix-vector multiplication can be implemented on a
hexagonal array. Problems of computing convolution, and finite impulse response (FIR) filters are such examples (Mead and Conway, 1980).

Some non-numerical problems can be implemented on the VLSI networks of processors when appropriate interpretations are given to the addition (+) and multiplication (×) operations. For example, some pattern-matching problems can be viewed as matrix problems with comparison and Boolean operations.

In conclusion, the inter-processor communications are still dominating the cost of parallel algorithms and systems. In fact, communication paths inherently take more space and energy than processing elements in many problems. For this reason an optimal choice of processors should be considered in the problem's implementation. However, for all the above implementations it appears that the optimal choice of the size of the network to solve a particular problem depends not only on the problem but also on the memory bandwidth to the host computer. For achieving high performance, it is desirable to have as many processors as possible in the network, provided they can be kept busy doing useful computations.

To this end, we conclude that the preceding sections discussed synchronous and asynchronous algorithms which are quite different. For synchronous parallel computers, one is concerned with algorithms defined on networks. The tasks of an algorithm are computations associated with the nodes of the underlying network. However, communication geometry and data movement are a major part of an algorithm in this computer. For the VLSI implementation, it is essential that the communication geometry be simple and regular, and that silicon area rather than the number of gates alone should be taken into consideration.

For asynchronous parallel computers, one is concerned with parallel algorithms whose tasks are executed by asynchronous processes. Such
processes require critical sections to protect shared data accessed by
more than one process, and the execution time for each process is random.
In general, when the fluctuations in computation time are large,
asynchronous algorithms are more efficient than synchronised versions for
the following reasons. First, the processes never waste time in waiting
for inputs. Second, the algorithms can take advantage of processes which
are run fast. The 'slow' processes which are doing useless computations
may be discovered and aborted at early times. Third, the algorithms are
"adaptive", so that the processes can finish at about the same time. This
guarantees that the maximum parallelism is used during most computation
times. Furthermore, asynchronous algorithms are more reliable than
synchronised algorithms in the sense that processes continue computing the
solution of their problems as long as no blocking occurs in the critical
sections, which are presumably small, although if some processes are
blocked forever.

In most of the examples discussed in this chapter a considerable
speed-up was achieved as well as high efficiency. In the analysis of any
parallel algorithm, the overheads are important and should be included in
the calculation. For example, for analysing an asynchronous algorithm it
is crucial to include overheads due to the execution of synchronisation
primitives and critical sections. Thus, it is important to determine the
number of processes in any algorithm to allow minimum communication over-
heads, hence a possible maximum speed-up. The analysis of the algorithms
presented in this thesis are concerned mainly on these basic concepts
which are crucial in parallel processing studies.
CHAPTER 4

THE DESIGN AND ANALYSIS OF ASYNCHRONOUS ALGORITHMS

FOR LINEAR EQUATIONS
4.1 INTRODUCTION

The work in this thesis is mainly concerned with asynchronous aspects of the MIMD computer. In this chapter, asynchronous numerical methods are presented. These methods employ asynchronous processes which are suitable for the MIMD computer. The motivation for this type of parallel algorithms stems from the fact that no parallel algorithm will load all the processors equally. It would then seem reasonable to look for algorithms which work on different processors without any synchronisation, i.e. asynchronous algorithms. These algorithms have achieved importance in parallel computing because the coordination of processors is an overhead in the execution of the algorithms, some algorithm/system combinations destroy any gains from having multiple processors co-operating on the algorithm execution, which means a reduction in the parallelism and decreases the maximum speed-up which one would expect to achieve in using a multiprocessor.

A wide variety of problems in numerical mathematics can be solved by solving sets of linear equations. For example, the computer solution of problems which have continuous analytic solutions and entails the replacement of the problem by a discrete or a series of discrete approximations. In particular, the numerical solution of partial differential equations that is required in the solution of the weather-forecasting problem, is accomplished by replacing the equations by a banded, system of linear equations whose solution yields an approximation to the exact solution in the form of a set of values generated by a function which approximates the true analytic solution. In order to obtain the required accuracy, it may be necessary to solve a set of linear equations in which the right hand side has been altered slightly. Also, it is expedient to compute the solution of the new set of equations from that of the old set by an iterative process, such as the Jacobi, Gauss-Seidel and Successive Over-
relaxation methods (see Section 4.3). The coefficient matrix for the set of linear equations is factorised in a way, which will be described later, from which different right hand sides may be generated to form the above mentioned methods. The factors obtained from the coefficient matrix are generated by using a "decomposition procedure".

The asynchronous form of the above-mentioned iterative methods have been developed by Baudet [1978a] where his studies were a consequence to the work presented by Kung [1976]. We have implemented these asynchronous algorithms on the NEPTUNE system and extended Baudet's work by implementing the Successive Over-relaxation (SOR) method and a block iterative method, particularly the four-point block method (see Chapter 5) to solve the Dirichlet problem of the Laplace equation where some numerical results are obtained and compared. Also, the Jacobi over-relaxation (JOR) method is implemented using a special type of matrix.

Algorithms with no synchronisation at all are called Purely Asynchronous (PA) (Baudet [1978a]). These algorithms have also been implemented and extended to the Purely Asynchronous Over-relaxation method (PAOR). The numerical results are obtained for all methods and compared and a performance analysis is presented to exploit the efficiency of the NEPTUNE system when these methods are implemented.

The theory of convergence for all these iterative methods is presented and the rules for obtaining an optimum relaxation factor (ω) for the successive over-relaxation or under-relaxation iterative methods are studied.

Finally, the behaviour of these asynchronous algorithms is analysed by Queueing theory concepts.
4.2 BASIC DEFINITIONS

As this chapter is designed mainly for numerical parallel implementations, therefore, some basic notations and definitions of matrix problems will be presented as well as some definitions that are relevant to the convergence theory and the rate of convergence of the iterative methods.

Matrix Properties

An \((n \times n)\) matrix, \(A\), denoted by \([a_{i,j}]\), for \(i,j=1,2,\ldots,n\) (see equation (3.2.1)), is a diagonal matrix if all its elements are zero except the main diagonal, i.e., the line represented by the set of elements \(a_{ii}\), for \(i=1,2,\ldots,n\). The diagonal matrix with \(a_{ii}=1\), for \(i=1,2,\ldots,n\) is called the identity matrix and is usually denoted by \(I\).

The matrix \(A=[a_{i,j}]\) is diagonally dominant if

\[
|a_{1,1}| \geq \sum_{j=1, j \neq 1}^{n} |a_{1,j}| , \text{ for all } 1 \leq i \leq n, \tag{4.2.1}
\]

In other words, each entry of the main diagonal is greater or equal to the sum of all other entries of the row holding that particular entry. \(A\) is said to be strictly diagonally dominant if strict equality holds for all \(1 \leq i \leq n\) in the equality (4.2.1).

The matrix \(A=[a_{i,j}]\) is a lower triangular matrix if \(a_{i,j}=0\) for all \(i<j\). Whereas, the matrix \(A\) is an upper triangular matrix if \(a_{i,j}=0\) for all \(i>j\).

The matrix \(A=[a_{i,j}]\) is said to be a band matrix if \(a_{i,j}=0\) for \(|i-j|>m\) (i.e. "bandwidth" \(2m+1\) since this is the number of non-zero diagonals in the band). For example, if \(a_{i,j}=0\) for \(|i-j|>1\), i.e. all elements are zero except for the main diagonal and sub and super diagonals, then the matrix \(A\) is said to be tridiagonal. In general, if there are \(m_1\) non-zero diagonals immediately below and \(m_2\) non-zero diagonals immediately above the main diagonal, then \(a_{i,j}=0\) for \(i>j+m_1\) and \(j>i+m_2\), and it follows that the matrix \(A\) is a band matrix of "bandwidth \(p=m_1+m_2+1\)."
If most of the elements $a_{i,j}$ of a matrix $A$ are zero then $A$ is said to be a sparse matrix. However, if most of the elements $a_{i,j}$ are non-zero, then the matrix $A$ is a dense matrix.

Let the determinant of a matrix $A$ be denoted by $\det(A)$. Thus, a matrix $A$ is non-singular if and only if $\det(A) \neq 0$.

An $(n \times n)$ matrix $A$ is irreducible if $n=1$ or if $n>1$ and given any two non-empty disjoint subsets $S$ and $T$ of $W$, the set of the first $n$ positive integers, such that $S+T=W$, there exists $i \in S$ and $j \in T$ such that $a_{i,j} \neq 0$ (Young, 1971).

A matrix $A$ of order $n$ has property (A) if there exist two disjoint subsets $S$ and $T$ of the first $n$ positive integers $W$ such that $S+T=W$ and such that if $i \neq j$ and if either $a_{i,j} \neq 0$ or $a_{j,i} \neq 0$ then $i \in S$ and $j \in T$ or else $i \in T$ and $j \in S$ (Young, 1971).

An immediate consequence of this definition is that a matrix $A$ which has property (A) can, by suitable permutation of its rows and columns, be written as

$$
\begin{bmatrix}
D_1 & F \\
G & D_2
\end{bmatrix}
$$

where $D_1$ and $D_2$ are square diagonal matrices.

A matrix $A$ of order $n$ is consistently ordered if for some $t$ there exist disjoint subsets $S_1, S_2, \ldots, S_t$ of $W=\{1,2,\ldots,n\}$ such that $\sum_{k=1}^{t} S_k = W$ and such that if $i$ and $j$ are associated, then $j \in S_{k+1}$ if $j>i$ and $j \in S_{k-1}$ if $j<i$, where $S_k$ is the subset containing $i$ (Young, 1971).

### Eigenvalues and Eigenvectors

The eigenvalue problem of a given $(n \times n)$ matrix $A$ is to find the eigenvalues $\lambda$ and the eigenvectors $v (v \neq 0)$ such that,

$$Av = \lambda v$$

The eigenproblem may be written as:

$$(A-\lambda I)v = 0 \quad \text{(4.2.2)}$$
where \( I \) is the identity matrix. Equation (4.2.2) which is called the characteristic equation of the matrix \( A \), is also a system of \( n \) homogeneous linear equations. This system has a non-trivial solution, \( v \neq 0 \), if and only if the matrix of the system is singular, i.e.,
\[
\det(A - \lambda I) = 0.
\]

The characteristic equation (4.2.2) has \( n \) roots \( \lambda_i \), for \( i=1,2,\ldots,n \).

Also, there is at least one eigenvector solution \( v_k \) associated with each \( \lambda_k \) of
\[
(A - \lambda_k I)v_k = 0.
\]

Theoretically,
\[
\det(A) = \prod_{i=1}^{n} \lambda_i,
\]
and
\[
\text{trace}(A) = \sum_{i=1}^{n} \lambda_i.
\]

The spectral radius of a matrix \( A \) is defined as,
\[
\rho(A) = \max_{i \in \text{sign}} |\lambda_i|.
\]

A real matrix is positive definite if and only if it is symmetric and all its eigenvalues are strictly positive.

### Contracting Operators

Let \( F \) be an operator from \( \mathbb{R}^n \) to itself, i.e. \( F: \mathbb{R}^n \rightarrow \mathbb{R}^n \) that satisfies the relation \( x = F(x) \), where \( x \) is a vector in \( \mathbb{R}^n \). The following definitions will be applicable on \( F \).

**Definition 4.2.1**

An operator \( F: \mathbb{R}^n \rightarrow \mathbb{R}^n \) is a Lipschitzian operator on a subset \( D \) of \( \mathbb{R}^n \) if there exists a non-negative (\( n \times n \)) matrix \( A \) such that
\[
|F(x) - F(y)| \leq A|x - y|, \text{ for all } x, y \in D, \tag{4.2.4}
\]

where, if \( Z \in \mathbb{R}^n \) with components \( Z_i, i=1,2,\ldots,n \), \( |Z| \) denotes the vector with components \( |Z_i|, i=1,2,\ldots,n \), and the inequality holds for every component. The matrix \( A \) will be called a Lipschitzian matrix for the operator \( F \).
From this definition, one can see that any Lipchitzian operator is continuous and, in fact, uniformly continuous on D, (i.e. to satisfy the condition $F(a) = \lim_{x \to a} F(x)$).

**Definition 4.2.2**

An operator $F$ from $\mathbb{R}^n$ to $\mathbb{R}^n$ is a **contracting operator** on a subset $D$ of $\mathbb{R}^n$ if it is a Lipchitzian operator on $D$ with a Lipchitzian matrix $A$ such that $\rho(A) < 1$, (where $\rho(A)$ is the spectral radius of $A$). The matrix $A$ will be called a contracting matrix for the operator $F$.

The contracting operators are guaranteed to have a unique **fixed point** in the subset $D$ that can easily be derived from the definition. In addition, if we assume, for example, that $D$ is closed and that $F(D) \subseteq D$, we are also guaranteed of the existence of a fixed point in the subset $D$.

(Baudet, 1978a). Since, if we assume there are two fixed points $a$ and $b$ that satisfy the condition $a = \lim_{x \to c} F(x)$, $b = \lim_{x \to c} F(x)$, then by continuity, if $a < b$ then $F(a) < F(b)$

$$\Rightarrow F(\lim_{x \to c} F(x)) < F(\lim_{x \to c} F(x))$$

which is incorrect. Therefore, $a$ and $b$ must represent the same fixed point.
4.3 METHODS FOR SOLVING LINEAR SYSTEMS OF EQUATIONS

A finite difference method for approximating a boundary value problem leads to a system of algebraic simultaneous equations. For linear boundary value problems, these equations are always linear but their number is generally large and, for this reason, their solution is a major problem in itself.

These problems can be expressed in matrix-vector notation by the equation,

\[ Ax = b, \tag{4.3.1} \]

where \( A \) is an \((n \times n)\) matrix of the coefficients, \( b \) is a known \( n \)-vector and \( x \) is an unknown \( n \)-vector whose value is to be found.

Provided that \( \det(A) \) is non-zero the unique solution of the equation is expressed simply as

\[ x = A^{-1}b, \]

where \( A^{-1} \) is the inverse of the matrix \( A \).

However, in numerical practice it is uncommon to compute the inverse of \( A \) since more efficient ways of solving the problem are available. In addition, if the coefficient matrix is sparse, it is unlikely (except in very special cases) that the sparsity will be preserved in the inverse. However, methods of solution belong essentially to either the class of direct methods or the class of iterative methods.

4.3.1 Direct Methods

Direct methods solve the system of equations, equation (4.3.1), in a known number of arithmetic operations, and errors in the solution arise entirely from rounding errors introduced in the computation. A common direct method (a variant of the well known Gaussian elimination method) for the solution of (4.2.1) requires the decomposition of \( A \) into a pair of factors \( L \) and \( U \) where \( L \) is a lower triangular matrix and \( U \) is an upper triangular matrix of the same order as \( A \). This decomposition can only be
carried out provided that $A$ is non-singular. Equation (4.3.1) can then be replaced by,

$$LUX = b,$$

(4.3.2)

The solution of (4.3.1) is calculated from equation (4.3.2) by putting $UX = y$ and then solving $Ly = b$ for $y$ by forward substitution and $UX = y$ for $x$ by back-substitution.

When solving partial differential equations the coefficient matrix $A$ usually appears in a tridiagonal block structure which is preserved in the factors $L$ and $U$.

4.3.2 Iterative Methods

Iterative methods are most useful for improving approximate solutions which have been generated using a direct method. In any iterative method, the solution is obtained in which an initial approximation is used to calculate a second approximation which in turn is used to calculate a third and so on. The iterative procedure is said to be convergent when the difference between the exact solution and the successive approximations tend to zero as the number of iterations increases.

In these methods, it is useful to scale and arrange the equations in such a way that the matrix $A$ has the following splitting, i.e.

$$A = D - L - U,$$

where $L$ and $U$ are respectively lower and upper triangular matrices with null diagonals and $D$ is the diagonal matrix. Equation (4.3.1) may now be written as

$$(D-L-U)x = b,$$

(4.3.3)

which can be written as,

$$Dx = (L+U)x + b.$$

The *Jacobi iterative method* or the method of simultaneous corrections is defined as,

$$Dx^{(k+1)} = (L+U)x^{(k)} + b, \quad k \geq 0,$$

(4.3.4)

giving,

$$x^{(k+1)} = D^{-1}(L+U)x^{(k)} + D^{-1}b.$$

(4.3.5)
The matrix $D^{-1}(L+U)$ or $(I-D^{-1}A)$, where $I$ is the identity matrix, is called the point Jacobi iteration matrix. Each point $x_i$ for $i=1,2,...,n$ of the vector $x$ is then iterated as follows:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} + \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \quad k \geq 0,$$

(4.3.6)

Related to the Jacobi method is the Simultaneous Over-relaxation method (the JOR method). In this method the displacement vector, $d^{(k)} = x^{(k+1)} - x^{(k)}$, of the JOR method is taken to be $\omega$ times the displacement vector $d_1^{(k)}$ defined by the Jacobi iteration. Hence, by equation (4.3.4), we have,

$$Dd_1^{(k)} = D(x^{(k+1)} - x^{(k)}) = (L+U)x^{(k)} + b-Dx^{(k)},$$

hence the JOR iteration defined by,

$$d^{(k)} = \omega d_1^{(k)},$$

can be written as,

$$x^{(k+1)} - x^{(k)} = \omega D^{-1}(L+U)x^{(k)} + b - Dx^{(k)}.$$ 

Therefore,

$$x^{(k+1)} = \{\omega D^{-1}(L+U) + (1-\omega)\}x^{(k)} + \omega D^{-1}b, \quad k \geq 0,$$

(4.3.7)

where $\omega$ is the under-relaxation factor. If $\omega=1$ then we have the Jacobi method. Now, for $1 \leq i \leq n$, each component of the vector $x$ is represented as,

$$x_i^{(k+1)} = \omega \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} + \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) + (1-\omega)x_i^{(k)}, \quad k \geq 0,$$

(4.3.8)

In these two methods, the evaluation of one component in an iteration depends on the values of components that are only obtained from the previous iterations. This means that these methods are inherently sequential.

For most coefficient matrices the **Gauss-Seidel iterative method** converges more rapidly than the Jacobi matrix (Varga, 1962). The method is defined by the equation,

$$Dx^{(k+1)} = Lx^{(k+1)} + Ux^{(k)} + b,$$

(4.3.9)

which is written as,
Since $(D-L)$ is a non-singular matrix, equation (4.3.10) shows that the Gauss-Seidel point iteration matrix is $(D-L)^{-1}U$. From equation (4.3.9), the iteration of each point $x_i$ is given by:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left\{ b_i + \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} + \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right\}, \text{ for } i=1,2,...,n,$$

(4.3.11)

This iterative method has the computational advantage that it does not require the simultaneous storage of two approximations $x_i^{(k+1)}$ and $x_i^{(k)}$ as in the point Jacobi iterative method.

Related to the Gauss-Seidel iteration method is the Successive Over-Relaxation method (SOR method). In this method, the displacement or correction vector $d(k) = x^{(k+1)} - x^{(k)}$ of the SOR method is taken to be $\omega$ times the displacement vector $d(k)$ defined by the Gauss-Seidel iteration. Therefore, from equation (4.3.9), we have,

$$Dd(k) = D(x^{(k+1)} - x^{(k)}) = Lx^{(k+1)} + Ux^{(k)} - Dx^{(k)} + b.$$  

Hence, the SOR iteration defined by,

$$d(k) = \omega d(k)$$

is then obtained from,

$$x^{(k+1)} - x^{(k)} = \omega D^{-1}(Lx^{(k+1)} + Ux^{(k)} - Dx^{(k)} + b).$$

Therefore,

$$x^{(k+1)} = \omega D^{-1}Lx^{(k+1)} + [(1-\omega) + \omega D^{-1}U]x^{(k)} + \omega D^{-1}b. \quad (4.3.12)$$

Clearly, the choice of $\omega = 1$ yields the Gauss-Seidel method. From equation (4.3.12), the equation of each component $x_i$ can be formulated as:

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left\{ b_i + \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} + \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right\} - (\omega-1)x_i^{(k)}.$$

(4.3.13)

It was argued that the above methods in their original form required a form of synchronisation and hence are not considered suitable to implement on parallel computers. However, many studies on parallel computation have improved the above argument. For example, Stone [1973b] has factorized the
matrix of coefficients and calculated the corresponding components in each of the upper and lower triangular matrices. By using the ILLIAC IV, each processor is assigned to each component of the unknown vector. The processors are all working simultaneously, therefore, data communications could occur amongst the processors. The technique that Stone has used for solving a system of equations is called "recursive doubling" (see Chapter 3).

Clearly, using a large scale high speed computer leads to a significant increase in speed. However, the ILLIAC IV computer can perform $N$ computations simultaneously, where $N=64$ or one of its multiplicants. Hence, the ILLIAC IV is expected to perform the computations $N$ times faster than that on the sequential computer of the same inherent speed. However, we have to bear in mind the overheads obtained from the data communication among the processors which reduces the speed-up of the operation by a certain factor. Another work concerning this subject is that of Clint and Perrott [1980]. In their paper, some parallel algorithms are used to solve the Jacobi, Gauss-Seidel or SOR methods when they are implemented on a SIMD computer in which each processor executes one component of the vector $x$. Hence, all the components can be evaluated simultaneously at any one time.

Since we are concerned with a MIMD computer whose processors act asynchronously, therefore iterative methods for solving a system of linear equations have to be constructed in an asynchronous form, i.e. asynchronous iterative algorithms. By an asynchronous iterative algorithm we mean that the processes always use in any evaluation the values of the components that are currently available at the beginning of the computation. This means that each processor has to compute different and independent subsets of the components. The idea of performing iterative methods asynchronously is that each processor in one iteration can perform, for example, one or
more components of the vector $x$ using their initial values stored in a shared memory. In the next iteration, a processor requires to compute the same components making use of the values obtained from the previous iteration. Because of the hardware of the processors, not all of the processors are equal in their performance time. Therefore, each processor can use at any computation time the values of the vector components that are evaluated and released from the previous iteration. On the other hand, when the value of a component is not available, the processors can then use the current value that was used in the previous iteration as shown in Figure 4.1, where two processors are working asynchronously on a system of four equations.

**Figure 4.1:** Two processors working asynchronously on 4 equations

*The bar means: end of execution.*
4.4 MATHEMATICAL NOTATIONS FOR THE ASYNCHRONOUS ITERATIVE METHODS

This section represents a mathematical form for the asynchronous iterative methods that solve a system of equations.

Suppose that, $F$ is a linear operator from $\mathbb{R}^n$ to $\mathbb{R}^n$ and is given by

$$F(x) = Ax + b$$ (4.4.1)

where, $x$ is an unknown $n$-vector, $A$ is an $(n \times n)$ matrix of coefficient and $b$ is a constant $n$-vector.

Chazan and Miranker [1969] introduced the chaotic relaxation scheme, a class of iterative methods for solving the system of equations of equation (4.4.1). They also showed that iterations defined by a chaotic relaxation scheme converge to the solution of equation (4.4.1) if and only if $\rho(|A|) < 1$, where $\rho(|A|)$ is the spectral radius of the matrix $A$.

The main motivation of defining chaotic relaxation is that when iterative methods are implemented on a multiprocessor system by using a chaotic relaxation scheme, the communication and synchronisation between the co-operating processes are significantly reduced. This reduction is obtained by not forcing the processes to follow a predetermined sequence of computations, but by allowing a process to choose dynamically not only the components to be evaluated but also the values of the previous iterates used in the evaluation. However, this scheme has a restriction that there must exist a fixed positive integer $S$ such that, in carrying out the evaluation of the $i$th iterate, a process cannot make use of any value of the components of the $j$th iterate if $j < i - s$.

When each process evaluates a subset of the components of the solution vector, it may wait for the values of the other components that are carried out by the other processes to be released in order to use them in further iterations. Although, the amount of work for evaluating these components in each process is the same. In fact, the actual time of the computation differs slightly due to many reasons, such as the communication between the
processors, or the hardware design of each processor. However, this will require some form of synchronisation that one would wish to avoid. As the penalty factor (Kung [1976]) of synchronising the processes at the end of each iteration is very large, also the synchronisation primitives make the processors idle, therefore, the computational time which creates an unnecessary overhead increases accordingly. In consequence, the maximum speed-up that one expected to achieve in using a multiprocessor system may be reduced.

To avoid the above restriction of the chaotic relaxation, Baudet [1978a] introduced a class of asynchronous iterative methods in which the chaotic relaxation is considered as a special case. The mathematical form of the asynchronous iterative methods is given in the following definition.

Definition 4.4.1

Let $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ be a linear operator such that,

$$F = \begin{bmatrix} f_1(x) \\ \vdots \\ f_n(x) \end{bmatrix}, \quad X = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

Let $J = (J_j)_{j=1}^\infty$ be a sequence of non-empty subsets of $\{1, \ldots, n\}$ and let $S = (s_1(j))_{j=1}^\infty$ be a sequence of elements in $\mathbb{N}^n$.

Thus, a sequence, $X(j) = \begin{bmatrix} x_1(j) \\ \vdots \\ x_n(j) \end{bmatrix} \in \mathbb{R}^n$, for $1 \leq j < \infty$ is called an asynchronous iteration if the sequence $(X(j))_{j=1}^\infty$ is determined by a quadruple $(F, X(0), J, S)$ in the following way:

1. $F, J, S$ as defined above,

$$X(0) = \begin{bmatrix} x_1(0) \\ \vdots \\ x_n(0) \end{bmatrix}$$
The sequence \( x(j) \) defined by the asynchronous iteration results naturally from the successive approximation if it is performed on a MIMD computer without any synchronisation of the processors.

By equation (4.4.2), we mean that, to evaluate a component \( x_i \) in the \( j \)th iteration, we might consider its recent value obtained from the \((j-1)\)th iteration if it has been released. Otherwise, the previous values of the components obtained in the early iterations are to be considered for further evaluations.

It is obvious that an asynchronous iterative method is subject to use the most recent values of the components instead of the values of an early iterate, as it is stated in the second part of condition (4) above. However, the first part of condition (4) states that only components of the previous iterates can be used in the evaluation of a new iterate. Finally, condition (3) of the definition states that every component of the vector \( x \) is never abandoned during any iterate.

As an example of iterative methods, let us consider the point Jacobi method. Jacobi's method which is defined on the operator \( F \) and an initial vector \( x(0) \) is defined by:

\[
J_j = \{1, \ldots, n\}, \text{ for } j=1,2, \ldots \\
S_i(j) = j-1, \text{ for } j=1,2, \ldots, \text{ and } i=1, \ldots, n.
\]

This means, all the components \( x_1, \ldots, x_n \) will be evaluated at once by one computational process with the fact that the components of the new iterate, say \( j \), cannot be evaluated until the values of the components from the iterate \((j-1)\) have been obtained.
On the other hand, the *asynchronous Jacobi method*, is defined as

\[ J_j = \{1+(j-1 \mod n)\} \quad \text{for } j=1,2,\ldots \]

\[ S_{i,j} = n \cdot \lfloor (j-1)/n \rfloor \quad \text{for } j=1,2,\ldots \text{ and } i=1,\ldots,n. \]

This means that each component is evaluated by one process and up to \( n \) processes can be used to perform the computation. In this method, the computation of the components of the new iterate, \( j \) does not wait for the values of these components from the previous iterate, \( j-1 \). Instead, any recent available value of these components at that time is considered for that computation.
4.5 THE CONVERGENCE THEOREM OF ASYNCHRONOUS ITERATIVE METHODS

In the convergence theorem of iterative methods in general, the matrix of coefficients of a given problem must be required to satisfy stronger conditions. Before stating the convergence theorem of an asynchronous iteration which requires a sufficient condition only, we present some characteristics of a non-negative matrix and the spectral radius $\rho$.

Theorem 4.5.1 (Varga, 1962)

If $A$ is an $(n \times n)$ matrix then $A$ is convergent if and only if $\rho(A) < 1$.

The proof of this theorem is in (Varga, 1962).

Lemma 4.5.1

Let $A$ be a non-negative square matrix. Then $\rho(A) < 1$ if and only if there exists a positive scalar $\lambda$ and a positive vector $v$ such that:

$$Av \preceq \lambda v \text{ and } \lambda < 1. \quad (4.5.1)$$

Proof:

Let us assume that inequality (4.5.1) holds, then we want to prove that $\rho(A) < 1$. Inequality (4.5.1) can be written as:

$$||A||_v \preceq \lambda < 1,$$

where, the matrix norm $||A||_v$ is indicated by the vector norm defined by,

$$||x||_v = \max\{|x_i|/v_i \mid i=1,...,n\}.$$

Since,

$$||A|| = \sup_{x \neq 0} \frac{||Ax||}{||x||} \geq \rho(A) \quad (Varga, 1962),$$

then, it follows that the matrix $A$ is convergent and that implies $\rho(A) \preceq ||A|| \preceq \lambda < 1$, i.e. $\rho(A) < 1$.

Now, assume that $\rho(A) < 1$, and to prove $Av \preceq \lambda v$. Let $t$ be a non-negative small scalar and $A_t$ be the matrix obtained by adding $t$ to all null coefficients of $A$. Then, it is clear that, for any positive vector $x$, we have $Ax A_t x$. In particular, if $t=0$ then $A_0 = A$ and since $\rho(A) < 1$ by hypothesis, then, we try to keep $\rho(A_t) < 1$ by choosing $t > 0$ a very small quantity. In fact, also $\rho(A) \preceq \rho(A_t)$. 
By following the concepts of Perron's theorem (e.g. see Varga [1962]),
we let \( \lambda = \rho(A_t) \) and since \( A_t > 0 \) therefore, there exists a positive eigenvector \( v \) corresponding to the eigenvalue \( \lambda \).

Thus, the positive scalar \( \lambda \) and the positive vector \( v \) can verify the
\[
A_v \leq A_t v = \lambda v, \quad \text{with} \quad \lambda < 1.
\]
By this, we have completed the proof of both conditions, and now, we can state the convergence theorem and its proof.

Theorem (4.5.2) - The Convergence Theorem (Baudet [1978a])

If \( F \) is a contracting operator on a closed subset \( D \) of \( \mathbb{R}^n \) and if \( F(D) \subseteq D \), then any asynchronous iteration \( (F, X(0), J, S) \) corresponding to \( F \) and starting with a vector \( X(0) \) in \( D \) converges to the unique fixed point of \( F \) in \( D \).

Proof:
Let \( a \) be the unique fixed point of \( F \). We construct the operator as
\( F(x+a) = a \), and assume that \( F(a) = a = 0 \). If we let \( y = a \) in the inequality (4.2.4), we obtain,
\[
|F(x) - F(a)| \leq \lambda |x-a|,
\]
then
\[
|F(x)| \leq \lambda |x| \quad \forall \ x \in D.
\]
Let \( A \) be a contracting matrix for \( F \) and let \( \lambda \) and \( v \) be as defined in Lemma (4.5.1). Therefore, for any starting vector \( X(0) \) and since \( v \) is a positive vector, we find a positive scalar \( a \) such that \( |X(0)| \leq a v \).

We need to show that if we construct a sequence of indices \( j_p, p=0,1, \ldots \), such that the sequence of iterates of \( (F,X(0), J, S) \) satisfies:
\[
|X(j)| \leq a \lambda^p v, \quad \text{for} \ j \geq j_p \quad (4.5.2)
\]
Since \( 0 < \lambda < 1 \), then \( X(j) \rightarrow 0 \) as \( j \rightarrow \infty \) and this is what we want to prove.

First, suppose that \( p=0 \) and \( j_0 = 0 \), thus inequality (4.5.2) becomes
\[
|X(j)| \leq a v, \quad \forall \ j \geq 0 \quad (4.5.3)
\]
However, (4.5.3) is true for \( j=0 \) with a good choice of \( a \). Now, by hypothesis of the theorem, we consider \( X(k) \) for \( 0 \leq k \). Let \( Z \) denote a vector
of $\mathbb{R}^n$ with components $z_i = x_i(s_i(k))$, for $i=1,...,n$. From definition (4.4.1) of the asynchronous iterate, the components of $X(k)$ might be given by:

$$x_i(k) = x_i(k-1) \quad \text{if} \quad i \notin J_k,$$

in which case $|x_i(k)| = |x_i(k-1)| < a \nu_i$, or by

$$x_i(k) = f_i(z), \quad \text{if} \quad i \in J_k.$$

In this case, from condition 4 of definition (4.4.1) (i.e. $s_i(j) \leq j-1$) and since $s_i(k) < k$, then we have,

$$|F(z)| \leq A|z| \leq a \nu_i \leq a \nu \leq a \nu.$$

We have used here a forward substitution from Lemma (4.5.1). On the other hand,

$$|x_i(k)| = |f_i(z)| \leq a \nu_i,$$

and in this case, if $0 < \lambda < 1$, we can obtain $|x_i(k)| \leq a \nu_i$. Therefore, inequality (4.5.3) is proved by induction when $p=0$.

Now, we assume that $j_p$ has been found and that inequality (4.5.2) holds for $0 \leq p < q$. In fact, we want to find $j_q$ and show that inequality (4.5.2) is true for $p=q$.

Initially, we define $r$ by,

$$r = \min \{ k | \forall j \geq k \quad s_i(j) \geq j_{q-1}, \text{for} \quad i=1,2,...,n \}.$$

By our definition of asynchronous iterates, this number exists by the condition $s_i(j) \to \infty$ of definition (4.4.1) and by the condition $s_i(j) \leq j-1$ of the same definition, we have $r > j_{q-1}$, since $s_i(j) \geq j_{q-1}$ (from the definition of $r$) which yields $j_{q-1} \leq s_i(j) \leq j-1$ which implies $j_{q-1} < j-1$ or $r > j_{q-1}$.

Since we assumed that inequality (4.5.2) is true for any $p, 0 \leq p < q$, therefore,

$$|x(r)| \leq a \lambda^{q-1} \nu.$$

Take $j \geq r$ and consider the components $x(j)$. Let $Z$ be the vector with components $z_i = x_i(s_i(j))$. From the definition of $r$, we have $s_i(j) \geq j_{q-1}$, for $i=1,...,n$, and this shows that

$$|z_i| = |x_i(s_i(j))| \leq a \lambda^{q-1} \nu_i, \quad \text{or in general} \quad |z| = a \lambda^{q-1} \nu.$$
In particular, in using the contracting property of the operator $F$ we obtain:

$$|F(Z)| \leq A|Z| \leq A\lambda q^{-1}v \leq \alpha q^{-1}A\lambda v.$$  

Since $Av = \lambda v$ for $\lambda < 1$ (Lemma (4.5.1)) then, $|F(Z)| \leq \alpha q^{-1}Av \leq \alpha qv$.

This last inequality shows that, if $i \in J_j$, $x_i(j)$ satisfies $|x_i(j)| = |f_i(z)| \leq A\lambda q^{-1}v_i$.

On the other hand, if $i \not\in J_j$, the $ith$ component is not modified.

Therefore, as soon as the $ith$ component is updated between the $rth$ and the $jth$ iterations we have, $|x_i(j)| \leq A\lambda q^{-1}v_i$.

(4.5.4)

Now, we define $j_q$ as,

$$j_q = \min \{ j \mid j \geq r \text{ and } \{1, \ldots, n\} = J_r \cup \ldots \cup J_j \}.$$  

From condition (3) of definition (4.4.1), the number $j_q$ exists since the condition states that $i$ occurs infinitely many times in the subset $J_j$, $j=1,2,\ldots$. Thus, for any $j \geq j_q$ every component is updated at least once between the $rth$ and the $jth$ iteration since the union of $J_r, \ldots, J_j = \{1, \ldots, n\}$. Therefore, inequality (4.5.4) holds for $i=1, \ldots, n$. This shows that, for $p=q$, (4.5.4) also holds and this completes the proof of theorem (4.5.2).

Now, we consider some important measurements for the complexity of asynchronous iterative methods. Let $F: \mathbb{R}^n \rightarrow \mathbb{R}^n$ be an operator with a fixed point $\xi$ and satisfying the assumptions of Theorem (4.5.2). The measures of the efficiency for the convergence of the asynchronous iteration $(F, x(0), J, S)$ towards the fixed point $\xi$ of $F$ are described here.

In fact, if $F$ is a contracting operator with the contracting matrix $A$, we note that an estimate of the error committed with the asynchronous iteration $(F, x(0), J, S)$ is directly obtainable from the asynchronous iteration $(A, |x(0)-\xi|, J, S)$. However, $(A, |x(0)-\xi|, J, S)$ can reflect linear convergence only.

Throughout we only consider the convergence using the norm of the error vector $x(j)-\xi$ where the norm we choose is $||x|| = \max\{|x_i| | i=1,2,\ldots,n\}$,
this corresponds to the worst possible case for the convergence of the components.

To measure the linear convergence of the sequence of iterations \( x(j) \), \( j=1,2,... \) towards its limit \( \xi \), we consider the rate of convergence of the sequence which is defined as:

\[
R = \lim_{j \to \infty} \inf \frac{(-\log |x(j)-\xi|)}{j}.
\]  
(4.5.6)

This is the average rate of convergence (Varga [1962]).

In addition, if we let \( c_j \) be the cost associated with the first \( j \) iterates, i.e. \( x(1), x(2),..., x(j) \), we define the efficiency of the sequence by,

\[
E = \lim_{j \to \infty} \inf \frac{(-\log |x(j)-\xi|)}{c_j}.
\]  
(4.5.7)

If all logarithms are taken to the base 10, \( 1/R \) measures the asymptotic number of steps required to divide the error by a factor of 10, whereas \( 1/E \) measures the corresponding cost. It is clear that if \( c_j/j \) tends to some finite limit \( \tau \) which corresponds to the average cost per step then the efficiency is simply given by \( E = R/\tau \).

The costs \( c_j \), \( j=1,2,... \), can be chosen in many ways. For example, we consider the cost to correspond either to the number of evaluations of the operator \( F \), or to the time to perform the evaluation. In the first case, if each component is equally as hard to compute, the cost can be directly evaluated from the sequence \( J \) by considering \( c_j = (|J_1|+...+|J_j|)/n \), where \( |J_j| \) is the cardinality of the set \( J_j \), i.e. the number of components evaluated at the \( j \)th step of the iteration. However, in the second case, the cost is better suited to deal with parallel algorithms and can be evaluated through the classical tools of queueing theory.
4.6 **FINITE DIFFERENCE APPROXIMATIONS TO DERIVATIVES**

The asynchronous iterative methods which are implemented on the NEPTUNE system are used to solve two problems. The first problem was to consider the one dimensional Dirichlet problem for solving Laplace Equation which is one case of the elliptic partial differential equations which arise usually from equilibrium or steady-state problems and their solutions.

Elliptic equations are the Poisson's equation

\[
\frac{\partial^2 \phi}{\partial x^2} = f(x), \quad (4.6.1)
\]

and the Laplace's Equation

\[
\frac{\partial^2 \phi}{\partial x^2} = 0. \quad (4.6.2)
\]

Laplace's Equation arises in the theories associated with the steady flow of heat or electricity in homogeneous conductors, with the irrotational flow of incompressible fluid, and with potential problems in electricity, magnetism and gravitating matter at points devoid of these entities.

To illustrate the use of equation (4.6.1), for example, let us suppose that there is a line of points separated by equal distances of length \( h \), i.e.,

\[
\phi = a \quad \begin{array}{cccccc}
1 & 2 & 3 & \ldots & i-1 & i & i+1 & \ldots & n
\end{array} \quad \phi = b
\]

Therefore, by using a Taylor's series expansion we get, for all \( 1 \leq i \leq n \),

\[
\phi(x_i+h) = \phi_{i+1} = \phi_i + \frac{d\phi_i}{dx} + \frac{h^2}{2!} \frac{d^2\phi_i}{dx^2} + \frac{h^3}{3!} \frac{d^3\phi_i}{dx^3} + \ldots
\]

\[
\phi(x_i-h) = \phi_{i-1} = \phi_i - \frac{d\phi_i}{dx} + \frac{h^2}{2!} \frac{d^2\phi_i}{dx^2} - \frac{h^3}{3!} \frac{d^3\phi_i}{dx^3} + \ldots
\]

By adding these two equations together, we obtain:

\[
\phi_{i+1} + \phi_{i-1} - 2\phi_i = h^2 \frac{d^2\phi_i}{dx^2} + O(h^4),
\]

which implies,

\[
\frac{d^2\phi_i}{dx^2} = \frac{\phi_{i+1} + \phi_{i-1} - 2\phi_i}{h^2}.
\]
By applying equation (4.6.1) we get,$$
abla \phi_i = f_i h^2,$$
which is the computational molecule of the form:

$$\begin{array}{c|c|c|c|c|c|c|c|c}
1 & -\frac{1}{2} & -\frac{1}{2} & 0 & -\frac{1}{2} & \frac{1}{2} & \cdots & \frac{1}{2} & 1 \\
-\frac{1}{2} & 1 & -\frac{1}{2} & -\frac{1}{2} & 1 & -\frac{1}{2} & \cdots & \frac{1}{2} & -\frac{1}{2} \\
0 & -\frac{1}{2} & 1 & -\frac{1}{2} & -\frac{1}{2} & 1 & \cdots & \frac{1}{2} & 0 \\
\end{array}$$

FIGURE 4.2: The (n×n) matrix obtained from the one dimensional elliptic equation

where the vector \( B \) represents the boundary values of the problem. Moreover, the matrix \( A \) can be 'split' into lower and an upper triangular matrices to form different iterative methods as described in Section (4.3.2) from which we have some experimental results (see Section 4.9).

The second problem that was used in the implementation of the iterative
methods was the two-dimensional problem for the Laplace Equation,
\[
\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0,
\]
where \( \phi \) is a dependent variable applied to a connected region \( R \) in the \( x-y \) plane. When some lines are drawn parallel to the \( x \)-axis and lines parallel to the \( y \)-axis they intersect in points. These points are usually called the mesh points. Therefore, solving a set of these points yields an approximation of the partial differential equation.

Consider, now, a uniform grid of mesh size \( h \) on the unit square as shown in Figure 4.3.

\[\text{FIGURE 4.3: The grid of points in a unit square}\]

Assume that \( \phi(x,y) \) is differentiable, then by Taylor's series we have,
\[
\phi(x+h,y) = \phi(x,y) + h \frac{\partial \phi}{\partial x} + \frac{h^2}{2!} \frac{\partial^2 \phi}{\partial x^2} + \frac{h^3}{3!} \frac{\partial^3 \phi}{\partial x^3} + \ldots, \tag{4.6.4}
\]
\[
\phi(x,y+h) = \phi(x,y) + h \frac{\partial \phi}{\partial y} + \frac{h^2}{2!} \frac{\partial^2 \phi}{\partial y^2} + \frac{h^3}{3!} \frac{\partial^3 \phi}{\partial y^3} + \ldots, \tag{4.6.5}
\]
\[
\phi(x+h,y+h) = \phi(x,y) + h \left( \frac{\partial \phi}{\partial x} + \frac{\partial \phi}{\partial y} \right) + \frac{h^2}{2!} \left( \frac{\partial^2 \phi}{\partial x^2} + 2 \frac{\partial^2 \phi}{\partial x \partial y} + \frac{\partial^2 \phi}{\partial y^2} \right) + \frac{h^3}{3!} \left( \frac{\partial^3 \phi}{\partial x^3} + 3 \frac{\partial^3 \phi}{\partial x^2 \partial y} + 3 \frac{\partial^3 \phi}{\partial x \partial y^2} + \frac{\partial^3 \phi}{\partial y^3} \right) + \ldots, \tag{4.6.6}
\]
and
\[
\phi(x-h,y+h) = \phi(x,y) - h \left( \frac{\partial \phi}{\partial x} + \frac{\partial \phi}{\partial y} \right) + \frac{h^2}{2!} \left( \frac{\partial^2 \phi}{\partial x^2} - 2 \frac{\partial^2 \phi}{\partial x \partial y} + \frac{\partial^2 \phi}{\partial y^2} \right) - \frac{h^3}{3!} \left( \frac{\partial^3 \phi}{\partial x^3} - 3 \frac{\partial^3 \phi}{\partial x^2 \partial y} + 3 \frac{\partial^3 \phi}{\partial x \partial y^2} - \frac{\partial^3 \phi}{\partial y^3} \right) + \ldots, \tag{4.6.7}
\]
where the points \((x \pm h, y), (x, y \pm h)\) and \((x \pm h, y \pm h)\) are contained in \(R\). Now, if we combine equations (4.6.4) and (4.6.5) together, we obtain approximations to the second-order derivatives,

\[
\frac{\partial^2 \phi}{\partial x^2} = \frac{\phi(x+h,y) - 2\phi(x,y) + \phi(x-h,y)}{h^2} + o(h^2), \tag{4.6.8}
\]

\[
\frac{\partial^2 \phi}{\partial y^2} = \frac{\phi(x,y+h) - 2\phi(x,y) + \phi(x,y-h)}{h^2} + o(h^2). \tag{4.6.9}
\]

A combination of equations (4.6.6) and (4.6.7) gives an approximation to the second-order mixed derivative,

\[
\frac{\partial^2 \phi}{\partial x \partial y} = \frac{\phi(x+h,y+h) - \phi(x+h,y-h) - \phi(x-h,y+h) + \phi(x-h,y-h)}{4h^2} + o(h^2). \tag{4.6.10}
\]

If for the mesh point \((x_i, y_j)\) = \((ih, jh)\), we denote,

\[
\phi(x_i, y_j) \text{ by } \phi_{i,j}, \text{ then Laplace's Equation}
\]

\[
\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0,
\]

can be replaced at the point \((x_i, y_j)\) by the finite difference equation which is obtained from adding equations (4.6.8) and (4.6.9), thus,

\[
\frac{1}{h^2} (\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - 4\phi_{i,j}) = 0. \tag{4.6.11}
\]

Multiply by \(-h^2\) we obtain,

\[
4\phi_{i,j} - \phi_{i+1,j} - \phi_{i-1,j} - \phi_{i,j+1} - \phi_{i,j-1} = 0, \tag{4.6.11'}
\]

which is known as the 5-point finite difference equation.

From equation (4.6.11), we obtain a set of simultaneous equations whose solution is a finite-difference approximation of the exact solution \(\{\phi_{i,j}\}\) at the internal mesh points. In other words, the evaluation of any point \(\phi_{i,j}\) is related to the values of its nearest neighbours. However, equation (4.6.11) corresponds to the computational molecule,
Therefore, the \((16 \times 16)\) matrix illustrated below represents the coefficient matrix which is derived when a second order elliptic partial differential equation (i.e. the Laplace equation) is discretised on a network of lines spaced 1/5 apart, thus,

\[
\begin{array}{cccc}
4 & 8 & 12 & 16 \\
3 & 7 & 11 & 15 \\
2 & 6 & 10 & 14 \\
1 & 5 & 9 & 13 \\
\end{array}
\]

Thus, using the computational molecule above, we obtain the following \((16 \times 16)\) matrix (Figure 4.4).

\[
\begin{array}{cccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
1 & 4 & -1 & & & & & & & & & & & & & \\
2 & -1 & 4 & -1 & & & & & & & & & & & & & \\
3 & -1 & 4 & -1 & & & & & & & & & & & & & \\
4 & & & & & & & & & & & & & & & \\
5 & & & & & & & & & & & & & & & \\
6 & & & & & & & & & & & & & & & \\
7 & & & & & & & & & & & & & & & \\
8 & & & & & & & & & & & & & & & \\
9 & & & & & & & & & & & & & & & \\
10 & & & & & & & & & & & & & & & \\
11 & & & & & & & & & & & & & & & \\
12 & & & & & & & & & & & & & & & \\
13 & & & & & & & & & & & & & & & \\
14 & & & & & & & & & & & & & & & \\
15 & & & & & & & & & & & & & & & \\
16 & & & & & & & & & & & & & & & \\
\end{array}
\]

**FIGURE 4.4**

In this matrix, each row contains at most five non-zero entries, i.e. the matrix is sparse so that the computation of any iterative method concentrates only on these five points in each row.

This completes the brief descriptions of the one- and two-dimensional Laplace Equations on which our study is based in the next sections.
4.7 THEORETICAL DETERMINATION OF THE OPTIMUM RELAXATION FACTOR FOR ITERATIVE METHODS

The JOR and SOR iterative methods described in Section (4.3.2) show a reasonable improvement over the Jacobi and the Gauss-Seidel methods respectively if the relaxation factor $\omega$ is well chosen. However, the definition of the asynchronous iterative methods can be extended by introducing a relaxation factor $\omega > 0$ (Baudet [1978a]). Therefore, the operator $F$ of definition (4.2.2) can be represented as:

$$F_\omega = \omega F + (1-\omega)E,$$

where $E$ is the identity operator of $R^n$. It follows that,

$$|F_\omega(x) - F_\omega(y)| \leq \omega |F(x) - F(y)| + (1-\omega)|x-y|.$$ 

To prove $F_\omega$ is a contracting operator with a contracting matrix $A$, $F_\omega$ is a Lipchitzian operator with the Lipchitzian matrix,

$$A_\omega = \omega A + |1-\omega| I.$$ 

$A_\omega$ is the iteration matrix. Since $A$ is a non-negative matrix with $\rho(A) < 1$ and we have,

$$\rho(A_\omega) = \omega \rho(A) + |1-\omega|,$$ 

(4.7.1)

then, the range of $\omega$ that makes $\rho(A_\omega) < 1$ can now be obtained. As it is known that the spectral radius of a matrix is the maximum absolute eigenvalue of the matrix, therefore, to show, $|\rho(A_\omega)| < 1$ we write,

$$-1 < \rho(A_\omega) < 1,$$

then,

$$-1 < \omega \rho(A) + \omega - 1 < 1$$

by adding 1 to the equality, we obtain,

$$0 < \omega \rho(A) + \omega < 2$$

$$0 < \omega (\rho(A) + 1) < 2$$

then,

$$0 < \omega < \frac{2}{1+\rho(A)}.$$ 

(4.7.2)

Equality (4.7.2) gives the range of $\omega$ by which the method converges. This indicates that $F_\omega$ becomes a contracting operator.
From the above range of \( \omega \), it is known that the maximum value \( \omega \) can have is \( \frac{2}{1+\rho(A)} \), so that when \( \rho(A) \) is slightly less than 1, \( \omega \) becomes slightly greater than one and this implies that the problem to be solved cannot be very much accelerated, i.e. the model problem is not a very good example to illustrate this method.

Given a system of equations \( Ax=b \), we want to derive the optimum \( \omega \) for the JOR method. Let \( B=I-D^{-1}A \) denote the Jacobi iteration matrix and let \( m(B) \) and \( M(B) \) denote the minimum and the maximum eigenvalues of \( B \). Therefore,

\[
\rho(B) = \max( |m(B)|, |M(B)| ).
\]

We remark that if the contracting matrix \( A \) has property (A), then the eigenvalues of \( B \) satisfy

\[ m(B) = -M(B) \]  
(Young and Hageman [1981]) .

However, the optimum extrapolation method based on the Jacobi method, i.e. JOR, is always convergent. When the eigenvalues of \( B \) satisfy equation (4.7.3), \( \omega=1 \). For this case, the JOR method reduces to the Jacobi method without extrapolation. Since \( B = \omega B + (1-\omega)I \) represents the JOR iteration matrix, then in general, the optimum \( \omega \) occurs when the minimum and the maximum eigenvalues of \( B \) are equal in their values and opposite in their sign.

Thus, from equation (4.7.1) we have,

\[
m(B) = \omega m(B) + 1-\omega,
\]
and

\[
M(B) = \omega M(B) + 1-\omega.
\]

Hence, the optimum \( \omega \) occurs when,

\[
m(B) = -M(B), \text{ that is,}
\]
\[
\omega m(B) + 1-\omega = -\omega M(B) - 1 + \omega
\]
\[
2 = 2\omega - \omega m(B) - \omega M(B),
\]

which implies that,

\[
\omega = \frac{2}{2-m(B)-M(B)},
\]

which is the value of the optimum \( \omega \) for the JOR method. The diagram in Figure 4.5 illustrates the values of \( \omega \) and the optimum \( \omega \).
For the SOR method we want to find the optimum value \( w_b \) of \( w \) which minimizes the spectral radius of the SOR iteration matrix and thereby maximizes the rate of convergence of the method. At the present time no formula for \( w_b \) for an arbitrary set of linear equations can be derived. However, it can be calculated for many of the difference equations approximating first- and second-order partial differential equations because their matrices are often of a special type, i.e., 2-cyclic matrices which possess Property (A). Young [1954] proved that when a 2-cyclic matrix is put into what he called a consistently ordered form \( A \), which can be done by a simple re-ordering of the rows and corresponding columns of the original 2-cyclic matrix, then the eigenvalues \( \lambda \) of the point SOR iteration matrix \( L \), associated with \( A \) are related to the eigenvalues \( \mu \) of the point Jacobi iteration matrix \( B \) associated with \( A \) by the equation,

\[
(\lambda + w - 1)^2 = \lambda \omega^2 \mu^2 .
\]  

(4.7.5)

From this equation it can be proved that,
where \( \rho(B) \) is the spectral radius of the Jacobi iteration matrix.

To prove equation (4.7.6), we need to present some theorems and definitions.

It is sufficient to know that the block tridiagonal matrices \( A \) of the form,

\[
A = \begin{bmatrix}
D_1 & A_1 & & & \\
B_1 & D_2 & A_2 & & \\
& B_2 & D_3 & A_3 & \\
& & \ddots & \ddots & \ddots \\
& & & B_{k-1} & D_k
\end{bmatrix}
\]  

are 2-cyclic and consistently ordered, where \( D_i, i=1,2,\ldots,k \) are square diagonal matrices not necessarily of the same order.

**Theorem (4.7.1) (Smith [1978])**

If the block tridiagonal matrix of the form in equation (4.7.7) is written as \( A=D-L-U \), where \( D, L \) and \( U \) are respectively the diagonal, lower and upper matrices and the matrix \( A(\alpha) = D-\alpha L-\alpha^{-1} U \), then

\[
\det(A(\alpha)) = \det(A) = 0
\]

This theorem also holds when each \( D_i \) is a full matrix because the diagonal blocks of \( A(\alpha) \) are independent of \( \alpha \).

**Theorem (4.7.2) (Smith [1978])**

The non-zero eigenvalues of the Jacobi iteration matrix corresponding to the matrix \( A \) in equation (4.7.7) occurs in pairs \( \pm \omega \).

**Theorem (4.7.3) (Smith [1978])**

Let \( L \) and \( B \) represent respectively the SOR and the Jacobi point.
iteration matrices corresponding to the block tridiagonal matrix $A$ of (4.7.7). Then, if $\lambda$ is a non-zero eigenvalue of $L_\omega$ and $\mu$ satisfies the relation,
\[
(\lambda + \omega - 1)^2 = \lambda \omega^2 \mu^2, \quad \omega \neq 0
\] (4.7.8)
then $\mu$ is an eigenvalue of $B$. Conversely, if $\mu$ is an eigenvalue of $B$ and satisfies (4.7.8) then $\lambda$ is an eigenvalue of $L_\omega$.

**Theorem (4.7.4)**

When the matrix $A$ is block tridiagonal of the form given in (4.7.7) with non-zero diagonal elements, and all the eigenvalues of the Jacobi iteration matrix $B$ associated with $A$ are real and are such that $0 < \rho(B) < 1$, then,
\[
\omega_b = \frac{2}{1 + \sqrt{1 - \rho^2(B)}},
\]
and
\[
\rho(L_\omega) = \omega_b - 1.
\] (4.7.9)
Furthermore, the SOR method applied to the equations $Ax = b$ converges for all $\omega$ in the range $0 < \omega < 2$.

**Proof**

We can write equation (4.7.5) as
\[
\frac{1}{\omega}(\lambda + \omega - 1) = \pm \lambda^b \mu,
\]
and let
\[
y_1(\lambda) = \frac{1}{\omega}(\lambda + \omega - 1) = \frac{1}{\omega} \lambda + 1 - \frac{1}{\omega},
\]
and
\[
y_2(\lambda) = \lambda^b \mu.
\]
Then, the pair of eigenvalues of $L_\omega$ that correspond to the pair of the non-zero eigenvalues $\pm \mu$ of $B$ are the $\lambda$ values of the points where the straight line,
\[
y_1 - 1 = \frac{1}{\omega}(\lambda - 1)
\]
intersects the parabola
\[
y_2^2 = \lambda \mu^2.
\]

The straight line passes through the point $(1,1)$ and its slope $\frac{1}{\omega}$ decreases as $\omega$ increases. From Figure 4.6 it is seen that the largest abscissae of the two points of intersections decreases with increasing $\omega$ until the line is a tangent to the parabola. The $\lambda$ values of the points of
intersection are the \( \lambda \) roots of (4.7.5) which can be rearranged as,

\[
\lambda^2 + 2\lambda \{ (\omega-1) - 4\omega^2 \mu^2 \} - (\omega-1)^2 = 0.
\]

To find the point of tangency,

\[
y^2 = \lambda \mu^2 + 2ydy = \mu^2 d\lambda,
\]

since \( \frac{dy}{d\lambda} \text{ slope} = \frac{1}{\omega} = \frac{\mu^2}{2y} \). Then, \( y = \frac{\mu^2}{\omega} \).

and if we substitute \( y \) in \( y_1 \) above we get,

\[
\lambda = 1 + \omega(y-1) = 1 + \omega (4\omega^2 \mu^2 - 1)
\]

\[
= -((\omega-1) + 4\omega^2 \mu^2)
\]

The substitution of the value of \( \lambda \) in the above equation gives,

\[
\{ ((\omega-1) - 4\omega^2 \mu^2 \}^2 - (\omega-1)^2 = 0
\]

\[
(\omega-1)^2 - 4\omega^2 \mu^2 = \pm (\omega-1)
\]

\[
-4\omega^2 \mu^2 = \pm(\omega-1) - (\omega-1)
\]

\[
-\omega^2 \mu^2 = \pm 2(\omega-1) - 2(\omega-1)
\]

Then, if we choose the + sign of the RHS of the last equation we get a value for \( \omega \) that makes the value of \( \lambda>1 \), i.e. a non-convergent case. Thus, we have to choose the negative sign. Then,

\[
-\omega^2 \mu^2 = -2\omega^2 - 2\omega + 2 = -4\omega + 4
\]
\[ \omega^2 \mu^2 = 4\omega - 4. \]

By adding \( \omega^2 \) to both sides we obtain,

\[ \omega^2 \mu^2 + \omega^2 = \omega^2 + 4\omega - 4. \]
\[ \omega^2 - 4\omega + 4 = \omega^2 - \omega \mu^2. \]
\[ (\omega - 2)^2 = \omega^2 (1 - \mu^2). \]
\[ (\omega - 2) = \pm \omega \sqrt{1 - \mu^2}. \]

By taking only the negative sign, then,

\[ \omega_b = \frac{2}{1 + \sqrt{1 - \mu^2}}. \]

Now, we can show that the spectral radius of \( L_\omega \) which is the maximum eigenvalue \( \lambda \) that occurs at the point where the parabola \( y^2 = \lambda \mu^2 \) and its tangent intersect. Since the value of \( \lambda \) at this point is given above as,

\[ \lambda = 1 - \omega + \omega \mu^2, \]

then, by substituting the value of \( \omega_b \), we obtain,

\[ \bar{\lambda} = 1 - \frac{2}{1 + \sqrt{1 - \mu^2}} + \frac{4}{(1 + \sqrt{1 - \mu^2})^2} \]
\[ = 1 + 2\sqrt{1 - \mu^2} + (1 - \mu^2) - 2 - 2\sqrt{1 - \mu^2} + 2\mu^2 \]
\[ = \frac{\mu^2}{(1 + \sqrt{1 - \mu^2})^2} \]
\[ = \frac{1 - \sqrt{1 - \mu^2}}{(1 + \sqrt{1 - \mu^2}) (1 + \sqrt{1 - \mu^2})} \]
\[ = \frac{-1}{1 + \sqrt{1 - \mu^2}}. \]

\[ \therefore \quad \bar{\lambda} = \omega_b - 1. \]

For all values of \( \omega > \omega_b \), the \( \lambda \) roots of equation (4.7.5) are complex, each with modulus \( |\lambda| = \omega - 1 \) which increases when \( \omega \) increases. Since the SOR method diverges for all \( |\lambda| > 1 \) therefore, the case \( \omega > 2 \) gives a divergent iteration. This means, the iteration is convergent for \( 0 < \omega < 2 \). Since the maximum rate of convergence of the SOR method is determined by the minimum value of the eigenvalue of largest modulus of \( L_\omega \), \( \lambda = \lambda_1 \), say, it follows that,
\[ \lambda_1 = \omega_{\lambda_1} - 1, \]

where,

\[ \omega_{\lambda_1} = \frac{2}{1 + \sqrt{1 - \mu^2}} \]

and \( \lambda_1 \) corresponds to \( \pm \mu_1 \). In other words, the optimum value of \( \omega \) for the maximum rate of convergence is given by,

\[ \omega_\omega = \frac{2}{1 + \sqrt{1 - \rho^2(B)}} \]

where \( \rho(B) \) is the spectral radius of the Jacobi iteration matrix.
4.8 RATE OF CONVERGENCE OF THE BASIC ITERATIVE METHODS

If we consider the solution of the system of equations

\[ Ax = b, \]

where \( b \) is a known vector in \( \mathbb{R}^n \) obtained from the boundary conditions of the Dirichlet problem for the Laplace equation on a square mesh of \( \mathbb{R}^2 \), we now present the rates of convergence of the basic iterative methods described in Section (4.3.2).

Consider, for example, the square region \( R: 0 < x < \pi, 0 < y < \pi \), when \( h = \frac{\pi}{n} \), where \( R \) is divided into \( n^2 \) square nets with \( N \times N = (n-1)^2 \) interior points to be evaluated by employing the 5-point formula of equation (4.6.11). Let \( A \) denote the square matrix formed from the L.H.S. of equation (4.6.11), and the \( N \) eigenvectors of \( A \) are denoted by \( x_{pq} \) (\( ih, jh \)) with corresponding positive eigenvalues, \( \lambda_{pq} \) for \( p, q = 1, 2, \ldots, n-1 \).

From the relation,

\[ Ax_{pq} = \lambda_{pq} x_{pq}, \tag{4.8.1} \]

we have,

\[ x_{pq} = \sin(pih) \sin(jh), \tag{4.8.2} \]

\[ \lambda_{pq} = 4h^{-2} \left[ \sin^2 \left( \frac{ph}{2} \right) + \sin^2 \left( \frac{qh}{2} \right) \right]. \tag{4.8.3} \]

Clearly, by inspection, we have,

\[ \min_{pq} \lambda_{pq} = \lambda_{1,1} = 8h^{-2} \sin^2 \left( \frac{h}{2} \right), \]

\[ = 8h^{-2} \left[ \left( \frac{h}{2} \right)^2 - \left( \frac{h}{2} \right)^3 \frac{1}{3!} + \left( \frac{h}{2} \right)^5 \frac{1}{5!} - \ldots \right]^2 \]

\[ = 8h^{-2} \left[ \frac{h^2}{4} - \left( \frac{h^3}{12} \right)^2 + \ldots \right] \]

\[ = 2 - \frac{h^4}{12} + \ldots \]

Thus,

\[ \lambda_{1,1} \to 2 \text{ as } h \to 0. \tag{4.8.4a} \]

Similarly,

\[ \max_{pq} \lambda_{pq} = \lambda_{n-1,n-1} = 8h^{-2} (1 - \sin^2 \left( \frac{h}{2} \right)) \approx \frac{8}{h^2} \text{ as } h \to 0. \tag{4.8.4b} \]

Therefore, the spectral radius of \( A \) is \( \lambda_{n-1,n-1} \).

First, we consider the rate of convergence of the Jacobi method which is controlled by the eigenvalues of the matrix

\[ B = D^{-1} (L+U), \]
Since \( A = D - L - U \),
then,
\[
B = D^{-1} (D-A)
\]
\[
= I - D^{-1} A.
\]  \[(4.8.5)\]

In our problem, \( D = \left(\frac{4}{h^2}\right) I \), therefore the Jacobi iterative matrix becomes,
\[
B = I - \left(\frac{h^2}{4}\right) A.
\]

As the eigenvalues of \( A \) are given by equation (4.8.3), hence, the eigenvalues \( \lambda_{pq} \) of \( B \) are:
\[
\lambda_{pq} = 1 - \frac{h^2}{4} \left[ 4h^{-2} \left( \sin^2 \left( \frac{ph}{2} \right) + \sin^2 \left( \frac{qh}{2} \right) \right) \right]
\]
\[
= 1 - \sin^2 \left( \frac{ph}{2} \right) \sin^2 \left( \frac{qh}{2} \right)
\]
\[
= \cos^2 \left( \frac{ph}{2} \right) - \sin^2 \left( \frac{qh}{2} \right).
\]

Since \( \cos^2 x = \frac{1}{2} + \frac{1}{2} \cos 2x \) and \( \sin^2 x = \frac{1}{2} - \frac{1}{2} \cos 2x \), then,
\[
\lambda_{pq} = \frac{1}{2} \left( \cos (ph) + \cos (qh) \right), \text{ for } p,q=1,2,\ldots,n-1.
\]

Consequently, the spectral radius of \( B \) is
\[
\lambda(B) = \max |\lambda_{pq}| = \lambda_{n-1,n-1}
\]
\[
= \frac{1}{2} (\cos (n-1)h + \cos (n-1)h) = \cos (n-1)h
\]

By substituting the value of \( n \) in this equation yields:
\[
\lambda(B) = \cos h
\]
\[
\therefore \lambda(B) = \cos h - \left(1 - \frac{h^2}{2}\right) \text{ as } h \to 0.
\]  \[(4.8.6)\]

Hence, the rate of convergence of the Jacobi method is given by,
\[
R(B) = -\log_{10} (\cos h)
\]
\[
= -\log_{10} \left(1 - \frac{h^2}{2}\right) = \frac{h^2}{2} + O(h^4).
\]  \[(4.8.7)\]

Thus, the rate of convergence of the Jacobi method is \( \frac{h^2}{2} \) which is rather slow for small values of \( h \).

Now, consider the JOR method with the corresponding matrix,
\[
B_\omega = \omega B + (1-\omega) I,
\]
where \( \omega \) is the relaxation factor.

Let \( \gamma_{pq} \) denote the eigenvalues of \( B_\omega \), where \( p,q = 1,2,\ldots,(n-1) \). Since the eigenvalues of \( B \) are given in equation (4.8.6), therefore, \( \gamma_{pq} \) are:
\[
\gamma_{pq} = \frac{1}{2} \omega (\cos (ph) + \cos (qh)) + (1-\omega),
\]
and the spectral radius of \( B_\omega \), is,

\[
\gamma(B_\omega) = \max \left| | \gamma_{pq} | \right| = \gamma_{n-1,n-1}
\]

\[
= \frac{1}{n} \omega \cos(n-1)h + \cos(n-1)h + (1-\omega)
\]

\[
= \omega \cos(n-1)h + (1-\omega)
\]

\[
= \omega \cos \left( \frac{\pi}{h} - 1 \right) h + (1-\omega)
\]

\[
= \omega \cos h + 1 - \omega
\]

\[
\therefore \quad \gamma(B_\omega) = \omega \cos h + (1-\omega) - 2 (1 - \frac{h^2}{2}) + (1-\omega) \text{ as } h \to 0
\]

The rate of convergence then becomes:

\[
R(B_\omega) = -\log(\omega \cos h + 1 - \omega) - \log(1 - \frac{h^2}{2}) = \omega \frac{h^2}{2} + O(h^4) \text{ as } h \to 0.
\]  

(4.8.8)

Therefore, the rate of convergence of the JOR method is \( \omega \) times faster than the Jacobi method. It follows that, unless \( \omega > 1 \) this method cannot be accelerated and it can be considered to be the same as the Jacobi method.

Consider, now, the Gauss-Seidel method which is controlled by the iteration matrix,

\[
L = (I-L)^{-1}U.
\]

For our model problem, the maximum eigenvalue of \( L \) is given by

\[
\rho(L) = \cos^2 h - 1 + h^2 \text{ as } h \to 0, \quad \text{(Young [1971])}.
\]

Consequently, the rate of convergence is

\[
R(L) = -\log_{10}(\cos^2 h) = -\log_{10}(1 - h^2)
\]

\[
\approx h^2 + O(h^4),
\]

(4.8.9)

which is twice as fast as that for the Jacobi method.

The rate of convergence of the SOR method can now be obtained where the SOR iteration matrix can be obtained after reformulating equation (4.3.12) which becomes,

\[
\mathbf{x}(k+1) = (I-\omega L)^{-1}(\omega U + (1-\omega)I)\mathbf{x}(k) + (I-\omega L)^{-1}\omega \mathbf{b}.
\]

Thus,

\[
L_\omega = (I-\omega L)^{-1}(\omega U + (1-\omega)I)
\]

becomes the SOR iteration matrix.
It was proved that the spectral radius of the SOR method is
\[ \rho(L_w) = \omega_b - 1, \] (see Theorem 4.7.4)
where \( \omega_b \) is the optimum over-relaxation factor which is given by (Varga, 1962):
\[ \omega_b = \frac{2}{1 + \sqrt{1 - \rho^2(B)}}, \]
where \( \rho(B) \) is the spectral radius of the J method. Thus,
\[ \rho(L_w) = \frac{2}{1 + \sqrt{1 - \rho^2(B)}} \cdot -1 = \frac{1 - \sqrt{1 - \rho^2(B)}}{1 + \sqrt{1 - \rho^2(B)}}. \]
Since \( \rho(B) = \cos h \) then,
\[ \rho(L_w) = \frac{1 - \sin(h)}{1 + \cos h} = \frac{\rho(B)}{1 + \sqrt{1 - \rho^2(B)}} = \frac{1 - \frac{h^3}{3!} + \frac{h^5}{5!} - \ldots}{1 + \frac{h^3}{3!} + \frac{h^5}{5!} - \ldots}, \]
thus,
\[ \rho(L_w) = \frac{1 - h}{1 + h}, \] as \( h \to 0. \)
Therefore,
\[ \rho(L_w) = -\log_{10} \left( \frac{1 - h}{1 + h} \right) = -\log_{10} (1 - h) + \log_{10} (1 + h) \]
\[ = -[h - \frac{(-h)^2}{2} + \frac{(-h)^3}{3} - \ldots] + [h - \frac{h^2}{2} + \frac{h^3}{3} - \ldots] \]
\[ = 2h + \frac{2h^3}{3} + \frac{2h^5}{5} + \ldots \]
\[ = 2h + O(h^3), \text{ as } h \to 0, \] (4.8.10)
which is larger than that of the Gauss-Seidel method by the factor \( 2/h. \)
To this end it becomes clear that the SOR method is faster than the Gauss-Seidel method which is in turn faster than the Jacobi method.
However, all the iterative methods have been implemented on the NEPTUNE asynchronous system and the results obtained clearly identify the differences of the rates of convergence.
4.9 ITERATIVE METHODS ON THE NEPTUNE ASYNCHRONOUS SYSTEM

The solution of the Dirichlet problem described above was first considered for a simulated multiprocessor system where the chaotic iterations strategy suggested by Chazan and Miranker [1969] was used (Rosenfeld and Driscoll [1969]). Baudet [1978a] has also implemented the iterative methods which we have studied to solve the Dirichlet problem and we have extended these results.

4.9.1 Experimental Results with the Two-Dimensional Dirichlet Problem

The iterative methods described below have been implemented on the NEPTUNE system to solve the two-dimensional Dirichlet problem on a square mesh of points. The approximate solution to this problem is found by solving a linear system of equations. For this problem, \( Ax = b \) is solved and \( A \) is a \((nxn)\) sparse matrix and \( b \) is the vector obtained from the boundary conditions. The spectral radius of the matrix \( A, \rho(A) \), should be less than 1 to guarantee that any asynchronous iterative method implemented to solve the problem converges (see Section 4.5).

It is described in the previous section and Figure 4.4 that if there are 16 mesh points to evaluate then a \((16x16)\) sparse matrix will be obtained using the Finite Difference method. Hence, we want to solve an \((NxN)\) matrix linear system in a computer of \( P \) processors. The natural way to evaluate these points (or components) is to allocate a fixed amount of work to each process. In this case, we allocate \( S = \frac{N}{P} \) lines (i.e. rows) of the matrix to each process. The number of processes is taken to be less or equal to the number of processors. The rows are allocated to processes so that the first \( S \) rows are assigned to process 1, the second \( S \) rows are assigned to process 2 and so on. However the rows are processed sequentially although the components within each row are treated pairwise. This strategy is termed sequential decomposition.
Each process iterates on its subset permanently, but it requires to read all its components at the beginning of the iterate, then it releases all the values of the components for the next iterate. However, the read and the release of the components should be maintained in critical sections and the data structure holding the obtained values should be accessed by all processes, i.e. it should be stored in the shared memory. As discussed earlier critical sections cannot be accessed by many processes at a time, therefore, the processes possibly complete their work in different times, although they start their work at almost the same time. When a process P is busy releasing the values of its components, other processes cannot use these values when they require them until the release is completed. This means that the components of process P will iterate using the old values of these related components. This, however, increases the number of iterations that are required to find the solution within a required accuracy.

This strategy is implemented for all the basic iterative methods discussed in Section (4.3.2), which are implemented asynchronously. However, a synchronous version of the Jacobi iterative method is implemented to be the basis for our comparisons of the asynchronous iterative methods and to show their effectiveness. The methods are defined as follows:

The Synchronous Jacobi Method (or the J Method)

The Jacobi iterative method of equation (4.3.6) above is implemented by using the five-point formula given in equation (4.6.11). This method is convergent since the spectral radius of the Jacobi matrix of equation (4.3.5) is less than 1 (Varga [1962]).

From equation (4.3.6) we see that all the components of an iterate are computed simultaneously using the values of the previous iterate, and hence, parallelism can be introduced easily. Since, each process is allocated to a subset of rows to evaluate, therefore, when a process completes its computation,
it must then halt and wait for the other processes to complete their computations and then start the evaluation of the next iterate. The program that implements this method is shown as Program A.1 in Appendix A. However, the iteration counter is placed before the parallel paths (processes) are generated and incremented only when all paths are synchronised and the required accuracy has been reached. The experiments are carried out for two mesh sizes. Firstly, for a mesh size (12x12) i.e. 144 interior points to evaluate, and with the error difference (i.e. epsilon = 10^{-5}), secondly; for a mesh size (24x24), i.e. 576 points to evaluate, with epsilon = 10^{-2}. The NEPTUNE timing results, number of iterations and the speed-up are listed in Table 4.1. However, these results are taken for any number of paths less than or equal to the number of processors which are always 4. The boundary values used were 100 on the first column with the other boundaries set to zero. The speed-up ratio is taken as the ratio of the timing results when only one path is generated and when k paths are generated.

<table>
<thead>
<tr>
<th>Mesh size (N×N)</th>
<th>Epsilon</th>
<th>No. of paths</th>
<th>Timing in seconds</th>
<th>No. of iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12×12)</td>
<td>10^{-5}</td>
<td>1</td>
<td>238.730</td>
<td>306</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>132.520</td>
<td>306</td>
<td>1.80</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>100.770</td>
<td>306</td>
<td>2.37</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>87.380</td>
<td>306</td>
<td>2.73</td>
</tr>
<tr>
<td>(24×24)</td>
<td>10^{-2}</td>
<td>1</td>
<td>498.280</td>
<td>162</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>269.080</td>
<td>&quot;</td>
<td>1.85</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>199.520</td>
<td>&quot;</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>169.340</td>
<td>&quot;</td>
<td>2.94</td>
</tr>
</tbody>
</table>

**TABLE 4.1:** The synchronous Jacobi iterative method to solve the 2-dimensional Laplace equation using sequential decomposition

The decomposition of the rows into subsets in the form described previously is called "sequential decomposition". Another decomposition strategy was
implemented so that all the subsets contain an equal number of rows arranged in the following manner. Each row index is separated from the next row index by a distance equal to the number of the generated paths. This means that if 4 is the number of paths that are generated and the mesh contains 12 rows, then each subset contains 3 rows such that the subset $S_i$ contains the rows indexed as follows:

$S_1$ contains 1, 5 and 9,

$S_2$ contains 2, 6 and 10,

$S_3$ contains 3, 7 and 11,

and $S_4$ contains 4, 8 and 12.

We call this strategy "non-sequential decomposition". The results of this strategy are listed in Table 4.2. Both of these decomposition strategies are consistently ordered (Section 4.2) since the matrix for the two-dimensional Laplace Equation using the 5-point formula has property (A), then this matrix is of 2-cyclic form. These properties are mostly required for the SOR method to converge.

The timing results for the sequential decomposition are greater than that for the non-sequential decomposition strategies and this is related to the fact that the components in one row are related to the components in the row neighbours i.e. above and below, which are evaluated by other processes.

<table>
<thead>
<tr>
<th>Mesh size (N x N)</th>
<th>Epsilon</th>
<th>No. of paths</th>
<th>Timing in seconds</th>
<th>No. of iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12 x 12)</td>
<td>$10^{-5}$</td>
<td>1</td>
<td>127.710</td>
<td>168</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>84.960</td>
<td>206</td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>70.350</td>
<td>232</td>
<td>1.81</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>65.370</td>
<td>252</td>
<td>1.95</td>
</tr>
</tbody>
</table>

**TABLE 4.2**: The synchronous Jacobi iterative method using non-sequential decomposition
This means, in each iteration, the processes are synchronised and the values of their components are released, therefore the new values are used by other processes as soon as they are released. Whereas, in the sequential decomposition, most of the related components are in one process which are evaluated sequentially within the process. In fact, only the boundaries of the subsets in the processes are related in which case the evaluation of such components can be performed in parallel as in the non-sequential decomposition. This is why the number of iterations is large as well.

On the other hand, if we notice the speed-up ratios in Tables 4.1 and 4.2, we see that the speed-up in the sequential decomposition is greater than that of the non-sequential decomposition. This is because, the components in the sequential strategy are iterated equally whatever the number of paths is, while in the non-sequential decomposition, the number of iterations increases when the number of paths increases and this is due to the subsets' communication and the fact that the components cannot always get the recent values of those related components which are evaluated by other processes. Therefore, we conclude that the sequential strategy can exploit parallelism more than the non-sequential strategy in the synchronous iterative method. The full synchronisation of the processes in each iteration step is a significant overhead in an asynchronous multiprocessor in general and also in the parallel implementation of the Jacobi iterative method. This overhead can be decreased by introducing the Asynchronous Jacobi iterative method (or the AJ method) which is a related Jacobi method but each process iterates on its subset and never waits for other processes to complete their work. When each process completes the evaluation of the components in its subset, it releases their values to the other processes by updating the corresponding components of the shared data structure (in our case this is an array), and then immediately after it starts a new iteration step on its
subset by using in the computation, the values of the components as they are known at the beginning of the re-evaluation. In the AJ method, critical sections are required to update the components at the end of each iteration step and to copy the values of the components required for the next iteration. This method is implemented in Program A.2 in Appendix A. In this method, the iteration counter is placed within each process (path), therefore each process has its own counter and a convergence test is used to test whether its subset has converged or not. A global convergence test is required to ensure that all the components of all subsets have converged to the required accuracy. This is implemented by allocating a "flag" on each process to be set when the subset of the process has converged completely, then a simple test for all flags of the other processes is carried out. The process terminates its computation when all the flags are found to be set, i.e. when all subsets of other processes have converged. The flags are stored in the shared memory so that they can be read by all processes. The experimental results are listed in Tables 4.3 and 4.4, where two different mesh sizes were used with different error distances, and also the same boundary conditions as in the J method. The two decomposition strategies of the rows of the mesh as described in the J method were also implemented for the AJ method.

From Tables 4.3 and 4.4, we notice that the sequential decomposition strategy is better than the non-sequential decomposition where the timing results in the former are less than the results in the latter. The speed-up ratios are also good and slightly less than a linear speed-up in the sequential case, whereas it is not linear when the number of processes increases in the non-sequential case.

The difference in the results appears from the fact that since the algorithms run asynchronously, then the non-sequential decomposition allocates non-sequential rows to processes and since the AJ method uses
<table>
<thead>
<tr>
<th>Mesh size (N x N)</th>
<th>Epsilon</th>
<th>No. of paths</th>
<th>Timing in seconds</th>
<th>No. of iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12x12)</td>
<td>10^-5</td>
<td>1</td>
<td>233.040</td>
<td>306</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>125.380</td>
<td>321</td>
<td>1.86</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>86.860</td>
<td>326</td>
<td>2.68</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>67.920</td>
<td>333</td>
<td>3.43</td>
</tr>
<tr>
<td>(24x24)</td>
<td>10^-2</td>
<td>1</td>
<td>470.270</td>
<td>162</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>248.380</td>
<td>164</td>
<td>1.89</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>174.650</td>
<td>164</td>
<td>2.69</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>134.830</td>
<td>165</td>
<td>3.49</td>
</tr>
</tbody>
</table>

**TABLE 4.3**: The AJ method using the sequential decomposition strategy

<table>
<thead>
<tr>
<th>Mesh size (N x N)</th>
<th>Epsilon</th>
<th>No. of paths</th>
<th>Timing in seconds</th>
<th>No. of iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12x12)</td>
<td>10^-5</td>
<td>1</td>
<td>233.650</td>
<td>306</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>148.250</td>
<td>375</td>
<td>1.58</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>104.620</td>
<td>378</td>
<td>2.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>79.910</td>
<td>377</td>
<td>2.92</td>
</tr>
<tr>
<td>(24x24)</td>
<td>10^-2</td>
<td>1</td>
<td>489.220</td>
<td>162</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>270.700</td>
<td>174</td>
<td>1.81</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>184.300</td>
<td>174</td>
<td>2.65</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>140.120</td>
<td>174</td>
<td>3.49</td>
</tr>
</tbody>
</table>

**TABLE 4.4**: The AJ method using the non-sequential decomposition strategy

values of the components on the neighbouring rows from the old iteration, therefore the greater the number of processes the less the probability of the components to obtain the values of their related components that are evaluated by other processes. This is because the processes do not usually complete their computation at the same time, and because of the critical sections within each process, other processes may wait for the values of the components as they are updated by other processes.
Another observation from the above two tables is the number of iterations which increases slightly, in the case of the sequential strategy, when the number of processes increases from 1 to 4. While, in the non-sequential strategy, a sharp increase in the number of iterations is noticed between the two runs when only one process is used and when 2 to 4 processes were in use. As the number of iterations is important in any iterative method, therefore, the sequential decomposition strategy is chosen for further discussion.

A method faster than the Jacobi iterative method is the Gauss-Seidel method which is twice as fast (see Section 4.7). This method is described below.

Asynchronous Gauss-Seidel's Method (The AGS Method)

This iterative method implements the Gauss-Seidel method shown in equations (4.3.10) and (4.3.11), which is the same as the AJ method except that the process uses new values for the components in its subset as soon as they are known for further evaluations in the same cycle. However, the process releases the values of its components at the end of its cycle, in which case the other processes can use them when they are required. The sequential decomposition strategy was implemented in Program A.3 in Appendix A, where the same boundary conditions, and convergence test were used. Also, the processes were iterating asynchronously. The experiments were carried out also for the non-sequential decomposition strategy and the results for both implementations are listed in Tables 4.5 and 4.6.

The results of this method indicate also that the sequential decomposition is better suited for the Gauss-Seidel method than the non-sequential decomposition. This is due to the way the components of the Gauss-Seidel method iterate. To calculate any component of the mesh using the five-point
formula requires values from the same iteration step for some neighbouring components. Therefore, when the non-sequential decomposition is used, the related components go into different processes. Thus, when more than one process is cooperating at the same time and when the subset's size is sufficiently large, the components in each row may not obtain the most recent values of the 'north' and the 'south' components which are situated in the other process and instead they obtain the old values and iterate on them until the new values are released. This, in fact, reduces the speed-up, also it slows the process down and reduces the efficiency of the processor which is carrying out this process and thus many iterations will be required to solve this system of equations more than the sequential algorithm. Another factor that reduces the efficiency is the number of critical sections which read all the components and release them at the end of the evaluation. However, if \( k \), the number of processes is 2 and each one is a large subset, then the update of the shared data structure of the corresponding components performed within the critical section will take a larger time than the case for smaller size. Therefore, the other process will wait longer to enter the critical section to update its own components, since the access to a critical section is made sequential.

On the other hand, if the mesh has 12 rows then each process allocates 6 rows. If we imagine that the two subsets form a circular list, then, in this list, the newer dependent rows are 5 rows apart. Thus, if two processors are used, one can be 4 times slower than the other even though there is no interference between them.

In the case of \( k=3 \), the subsets contain odd and even rows in the alternative form. Obviously, the sizes of the subsets are smaller than that of \( k=2 \). Also, any row in any process depends on the 'north' and the 'south' rows which are allocated to different processes and not the same process.
Therefore, if 3 processors of different speeds are cooperating, then the probability of the row to obtain the most recent values from the neighbouring rows will be low. However, if we arrange the subsets to be in a circular list, thus:
When 3 processors are cooperating then each row is 3 rows apart from the neighbouring rows, and in any cycle two processors can be slower than the third one. Therefore, a speed-up cannot be achieved. Similarly, when \( k = 4 \) and 4 processors are cooperating.

In the sequential decomposition, all neighbouring components which are related are computed sequentially in one subset, except for the rows on the boundary of the subset which have their related rows in different subsets that are carried out by other processes. In this case, the component can obtain the new values of most of its related components which are in the same subset. This, however, explains why the number of iterations is less than that of the non-sequential decomposition case when the parallel running time is compared with the sequential one (i.e. when \( k = 1 \)). A speed-up is achieved accordingly and this is slightly improved when a larger mesh size is used. The reason is that the potentiality of a parallel system is exploited when a large mesh problem is considered.

This asynchronous iterative method can be improved if it is implemented without using critical sections, i.e. the synchronisation overhead can be decreased. Such a method is called the Purely Asynchronous method.

**Purely Asynchronous Iterative Method (the PA Method)**

This method implements the Gauss-Seidel concept i.e. equation (4.3.11) but each process obtains the new values of its components by using the most recent values of the components and releases each new value immediately after its evaluation. In other words the method is implemented as in the AGS method, where each process allocates a subset of components, but when each component is evaluated it is released immediately without waiting for other components of the same subset to be evaluated. This update can be made directly to the corresponding component of the shared array with the use of
a critical section. Therefore, each process always obtains the most recent value of any component it requires in its computation which implies that each process will iterate as many times as the others.

The sequential decomposition strategy for this algorithm is implemented in Program A.4 in Appendix A, where all the conditions used in the previous programs are used here as well. The results of this strategy together with the non-sequential strategy are listed in Tables 4.7 and 4.8 respectively.

It is obvious, from the above two tables, that in this method the two decomposition strategies behave equally, so that the same timing results, the same number of iterations and the same speed-up are obtained. This comes from the nature of the method in updating the components. Therefore, it does not matter in which process the related components are computed as long as they are released for other processes. However, a linear speed-up is achieved in this method and this is because of the absence of critical sections that generate substantial overheads.

When a component is updated by a process a direct access to the shared memory to update the corresponding component is made which means that, in this case there is no need for a local array to hold the values, instead only the shared array is used in a process to read from and to update its components. Obviously, one can think of a multiple access to the same location in the shared data. For example, one process is busy updating a component in the shared array whilst the other is trying to read the value of this component to be used in its computation. But, since the processes are of different speeds, the probability of such occurrence could be very low. Also, because of the sparsity and the special form of the matrix associated with our system of equations, accesses to the global array by a given process will be mostly confined to accesses of components within its own subset and only a few accesses to the components in the two adjacent subsets.
<table>
<thead>
<tr>
<th>Mesh size (N×N)</th>
<th>Epsilon</th>
<th>No. of paths</th>
<th>Timing</th>
<th>No. of iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12×12)</td>
<td>10⁻⁵</td>
<td>1</td>
<td>117.870</td>
<td>162</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>59.730</td>
<td>163</td>
<td>1.97</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>39.690</td>
<td>163</td>
<td>2.97</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>29.840</td>
<td>163</td>
<td>3.95</td>
</tr>
<tr>
<td>(24×24)</td>
<td>10⁻²</td>
<td>1</td>
<td>310.900</td>
<td>107</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>160.900</td>
<td>108</td>
<td>1.93</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>104.710</td>
<td>107</td>
<td>2.97</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>79.200</td>
<td>107</td>
<td>3.93</td>
</tr>
</tbody>
</table>

**TABLE 4.7:** The PA method using the sequential decomposition

<table>
<thead>
<tr>
<th>Mesh size (N×N)</th>
<th>Epsilon</th>
<th>No. of paths</th>
<th>Timing</th>
<th>No. of iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12×12)</td>
<td>10⁻⁵</td>
<td>1</td>
<td>118.140</td>
<td>162</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>59.660</td>
<td>163</td>
<td>1.98</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>39.990</td>
<td>163</td>
<td>2.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>30.050</td>
<td>163</td>
<td>3.93</td>
</tr>
<tr>
<td>(24×24)</td>
<td>10⁻²</td>
<td>1</td>
<td>311.420</td>
<td>107</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>157.270</td>
<td>107</td>
<td>1.98</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>104.720</td>
<td>107</td>
<td>2.97</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>78.790</td>
<td>105</td>
<td>3.95</td>
</tr>
</tbody>
</table>

**TABLE 4.8:** The PA method using the non-sequential decomposition

On conclusion of all the above results, the PA method is considered to be the best amongst the others. This is due to the absolute absence of any critical section which implies less overhead, and from the space or memory point of view, the method uses only one data structure. Whereas, the AJ method implements critical sections as well as employs a larger space. This is because, it requires from each process not only a complete duplication of all components (as at the beginning of its cycle) but still another copy of the components in its own subset. Although, this is a major concern in
practice, the experimental results show useful comparisons between the AJ method and the J method. The AGS method requires only one copy of the components to be local to the processes in addition to the shared data structure that holds the results. However, both of the AJ and AGS methods require a critical section to read in all the components to be used in each process at the beginning of the cycle and a critical section to release all components evaluated by a process to be available to other processes at the end of each cycle of the iteration step.

Now, by considering the sequential decomposition strategy, the timing results, speed-up ratios and the number of iterations obtained for the four methods, the synchronous Jacobi's, AJ, AGS and the PA methods are illustrated in Figures 4.7, 4.8 and 4.9 respectively, where the (24x24) mesh size is chosen. Figure 4.8 illustrates that a full linear speed-up is achieved in the PA method and the AJ method achieves a better speed-up than the AGS although the AJ method is slower to converge than the latter method. This is due to the behaviour of the method on iterating the components. On the other hand, Figure 4.9 illustrates that the Jacobi and PA methods produced a constant number of iterations for any number of processes, while in the AJ method the number of iterations increases slightly with increase of the number of processes. The AGS method shows a sharp increase with the increase of the number of processes. However, when the error difference is small, the increase of the number of iterations becomes clearer as the results of the case of the (12x12) mesh size indicate (see Table 4.5). This increase of the number of iterations reduces the speed-up and explains that the AJ method can be introduced easily to a parallel system rather than the AGS method which is an inherently sequential method.

All these asynchronous methods converge successfully and give the correct answer for the given boundary conditions when the algorithms run sequentially or in parallel.
FIGURE 4.7
The Timing Results of the J, AJ, AGS, and PA Methods
Solving the Two-Dimensional Dirichlet Problem

FIGURE 4.8
The Speed-up Achieved by the J, AJ, AGS, and PA Methods
FIGURE 4.9
The Number of Iterations Required to Solve The System of Equations by The Jacobi, AJ, AGS, and PA Methods
4.9.2 Experimental Results with the One-Dimensional Dirichlet Problem

The iterative methods described in Section 4.9.1 above have also been implemented asynchronously to solve the one-dimensional Dirichlet problem applied to the Laplace Equation. Thus, each component of the unknown vector is updated by using equation (4.6.3'), which means that each component depends only on the values of the left and right neighbouring components of the vector.

The decomposition strategy of the components was such that the n-vector is partitioned into P subsets consisting of $m = \frac{n}{p}$ components each arranged in sequential order. The same way of updating and reading components as in the 2-dimensional Dirichlet problem was applied in the case of the Asynchronous Jacobi and Gauss-Seidel methods. Also, the same test of convergence to terminate the iteration was used.

The experiments were carried out for the Jacobi method, AJ, AGS and the PA methods, where the (36 x 36) matrix size was chosen i.e. 36 equations to solve, and the error difference was $10^{-5}$. The experimental results of all these methods when implemented on the NEPTUNE system are listed in Table 4.9 where 4 processors were used in all the cases of the number of paths.

The timing results, speed-up ratios, and the number of iterations taken from Table 4.9 are illustrated diagrammatically in Figures 4.10, 4.11, 4.12, respectively. Figure 4.11 illustrates that also the PA method is the best amongst the other methods since it gives a linear speed-up. However, the AJ method has a good speed-up, as noticed from the figures in Table 4.9, and they are significantly greater than that of the method that solves the two-dimensional Dirichlet problem (see Table 4.3). These speed-up ratios are less than that of the PA method and this is due to the existence of critical sections in the former method that degrade the performance of the parallel system. The AGS method results in speed-up ratios significantly
<table>
<thead>
<tr>
<th>Method</th>
<th>Matrix Size (N\times N)</th>
<th>Epsilon</th>
<th>No. of paths</th>
<th>Timing in seconds</th>
<th>No. of iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>(36\times 36)</td>
<td>$10^{-5}$</td>
<td>1</td>
<td>258.130</td>
<td>1512</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>156.620</td>
<td>1512</td>
<td>1.65</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>112.100</td>
<td>1462</td>
<td>2.30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>90.250</td>
<td>1406</td>
<td>2.86</td>
</tr>
<tr>
<td>AJ Method</td>
<td></td>
<td></td>
<td>1</td>
<td>249.150</td>
<td>1512</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>130.360</td>
<td>1549</td>
<td>1.91</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>87.940</td>
<td>1525</td>
<td>2.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>67.010</td>
<td>1499</td>
<td>3.72</td>
</tr>
<tr>
<td>AGS Method</td>
<td></td>
<td></td>
<td>1</td>
<td>140.670</td>
<td>853</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>78.500</td>
<td>935</td>
<td>1.79</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>54.700</td>
<td>953</td>
<td>2.57</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>42.570</td>
<td>958</td>
<td>3.30</td>
</tr>
<tr>
<td>FA Method</td>
<td></td>
<td></td>
<td>1</td>
<td>134.500</td>
<td>853</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>67.950</td>
<td>857</td>
<td>1.98</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>44.880</td>
<td>840</td>
<td>2.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>33.700</td>
<td>821</td>
<td>3.99</td>
</tr>
</tbody>
</table>

**TABLE 4.9:** Different asynchronous iterative methods solving the one-dimensional Dirichlet problem

less than the ratios obtained from the AJ method. This is because, the components of the AGS method do not always iterate on the new values because they are not yet available. Therefore, these components violate the original scheme of the Gauss-Seidel method.

Finally, the number of iterations is different in this application than that of the two-dimensional Dirichlet problem. We notice that the number of iterations slightly decreases as the number of paths increases. While in the previous problem, the number was the same for any number of paths. In the Jacobi method also, the number of iterations decreases as the number of paths increases. Here, the decrease is significant. However, in
the AJ method, the number of iterations slightly decreases when the number of paths increases whereas the iterations increase in the AGS method.

We saw that the behaviour of the number of iterations in the Jacobi, AJ and the PA methods is different than that of the same method that solves the two-dimensional Dirichlet problem, see Figures 4.12 and 4.9 for comparison. This difference, however, is only related to the problem to be solved. Thus, in the one-dimensional problem, each component is related to the left and right neighbouring components only, hence the dependency is within the same subset. Although, many processes are co-operating most of the components can always obtain the new values of their related components since they are within the same subset, except of course the components on the boundary of the subsets. These components are few, only one in each boundary. This does not delay the computation very much, and the cooperation between processes is not very significant. Therefore, the timing results increase as well as the number of iterations when these methods run in parallel, i.e. in more than one process. Whereas, in the two-dimensional Dirichlet problem, the boundary in each subset consists of a full line (row) of components that are to be carried out by another process. This, however, causes some delay in each process in receiving the new values when required. Therefore, the number of iterations increases when the number of cooperating processes increases in which case the communication through critical sections acts as a bottleneck that degrades the efficiency of the parallel computer.
4.10 ITERATIVE METHODS WITH OVER- AND UNDER-RELAXATION FACTORS

The asynchronous iterative methods discussed in the previous section with their impressive results can be accelerated if an over-relaxation or an under-relaxation factor is added to the system of equations to accelerate the convergence of these methods.

The JOR and SOR methods of equations (4.3.8) and (4.3.13) respectively are introduced in this section together with the Purely Asynchronous Over-Relaxation (PAOR) method that solves the two-dimensional Laplace Equation.

Experimentally, the PAOR method is implemented on the NEPTUNE system where the same equation (4.3.13) is applied but the values of the mesh components are obtained as described in the Purely Asynchronous (PA) method above.

The experiments were carried out to solve a mesh of size (12x12), i.e. 144 components with the same convergence test and boundary conditions as used in the previous asynchronous iterative methods. The results of this algorithm are listed in Table 4.10, where different numbers of processes are carried out by 4 processors always. The parameter \( w \) represents the over-relaxation factor that accelerates the method. In these experiments, only the sequential decomposition is implemented.

A linear speed-up is achieved for this method as in the PA method but the timing results of this method are less than the PA method. Figure 4.13 illustrates the difference in the timing results of these two methods when a (12x12) mesh size is implemented. This difference can be calculated from the theory of the rate of convergence of the Gauss-Seidel method and the SOR method, since the PA and PAOR methods implement the Gauss-Seidel equations for computing the mesh components as mentioned earlier.

The rate of convergence of the Gauss-Seidel method shown in equation
FIGURE 4.10
The Timing Results of The J, AJ, AGS, and PA Methods
Solving The One-Dimensional Dirichlet Problem

FIGURE 4.11
The Speed-up Achieved by The J, AJ, AGS, and PA Methods
FIGURE 4.12
The Number of Iterations Obtained From The Iterative Methods Solving The 1-Dimensional Dirichlet Problem
<table>
<thead>
<tr>
<th>Mesh size (N×N)</th>
<th>No. of processors</th>
<th>( \omega )</th>
<th>Time (seconds)</th>
<th>No. of iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12×12)</td>
<td>1</td>
<td>1.0</td>
<td>136.800</td>
<td>162</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>69.930</td>
<td>163</td>
<td>1.96</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.0</td>
<td>47.210</td>
<td>163</td>
<td>2.90</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>36.010</td>
<td>163</td>
<td>3.80</td>
</tr>
<tr>
<td>(12×12)</td>
<td>1</td>
<td>1.62</td>
<td>26.270</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.62</td>
<td>13.840</td>
<td>29</td>
<td>1.90</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.62</td>
<td>8.980</td>
<td>28</td>
<td>2.93</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.62</td>
<td>6.650</td>
<td>28</td>
<td>3.95</td>
</tr>
<tr>
<td>(16×16)</td>
<td>1</td>
<td>1.0</td>
<td>393.840</td>
<td>263</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>207.010</td>
<td>265</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>105.090</td>
<td>270</td>
<td>3.75</td>
</tr>
<tr>
<td>(16×16)</td>
<td>1</td>
<td>1.70</td>
<td>55.600</td>
<td>37</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.70</td>
<td>29.460</td>
<td>38</td>
<td>1.89</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.70</td>
<td>15.130</td>
<td>38</td>
<td>3.70</td>
</tr>
</tbody>
</table>

**TABLE 4.10**: The PAOR method using the sequential ordering
(4.8.9) is given as $h^2$, where $h = \frac{n}{13}$. Since our model problem's size $N=12$ then $n=13$ (see Section 4.8). If we let $R$ denote the rate of convergence, therefore,

$$R_{(PA)} \approx h^2 = \left(\frac{n}{13}\right)^2 = 0.0584.$$  

The rate of convergence of the PAOR method is given as the SOR rate of convergence as in equation (4.8.10), thus,

$$R_{(PAOR)} = 2h = 0.48332.$$  

Then, the ratio of $R_{(PAOR)}$ over the $R_{(PA)}$ gives the factor of improvement that is obtained. This factor is given by,

$$\frac{R_{(PAOR)}}{R_{(PA)}} = \frac{0.48332}{0.0584} \approx 8.3.$$  

This shows that there is a factor of improvement of about 8.3 for the PAOR method as compared with PA method. This factor increases as $h$ decreases, i.e. as the order of the system of equations to be solved increases. For example, if $N=32$ then the factor becomes 20.5.

Now, if we compare the experimental results of the PA and PAOR methods listed in Tables 4.7 and 4.10, respectively when a (12x12) mesh size is used, we see that the factor of improvement is about 5 when only the timing results for different numbers of paths are compared.

Thus, when,

<table>
<thead>
<tr>
<th>No. of paths</th>
<th>$\frac{117.870}{26.270}$</th>
<th>4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$\frac{59.730}{13.840}$</td>
<td>4.3</td>
</tr>
<tr>
<td>3</td>
<td>$\frac{39.690}{8.980}$</td>
<td>4.4</td>
</tr>
<tr>
<td>4</td>
<td>$\frac{29.840}{6.550}$</td>
<td>4.6</td>
</tr>
</tbody>
</table>

These figures show that the experimental results are not sufficiently adequate to explain the theoretical results drawn from the rate of convergence theory. This, in fact, is due to different factors which arise from the
parallel implementation such as the allocation of subsets to parallel paths, the paths cooperations, the assignment of processors to paths and the shared data access overheads. All these factors may decrease the factor of improvement from its theoretical counterpart.

On the other hand, the optimum $\omega$ that is obtained from the experiments is 1.62 for a (12x12) mesh size when different number of parallel paths up to the number of available processors were used. This optimum value is similar to the theoretical optimum $\omega$ which is obtained from equation (4.7.6) as follows:

$$\omega_b = \frac{2}{1 + \sqrt[2]{1 - \rho^2}} = \frac{2}{1 + \cos^2 \left(\frac{\pi}{13}\right)} = 1.614.$$  

The asynchronous behaviour of the algorithm does not affect the value of the optimum $\omega$ when it is run in parallel as some other algorithms do. A method with such effect is the AGS method implemented on a (12x12) mesh size with the over-relaxation factor $\omega$. We call this method the Asynchronous Successive Over-relaxation (ASOR) method. It is implemented to solve the two-dimensional Dirichlet problem using the five-point formula on a square mesh in the same manner as described for the AGS method except that each component is iterated according to the formula of equation (4.3.13).

Obviously, to solve the (12x12) mesh, i.e. to evaluate 144 components of the mesh, it is required to have an optimum $\omega$ as 1.62 as was described in the case of the PAOR method. The results of the ASOR algorithm are listed in Table 4.11, where $1.3 \leq \omega < 2$ and the number of paths as 1, 2, 3 and 4. When $\omega = 1.0$ the method reduces to the AGS method. Thus, this value of $\omega$ is included in the table to compare the effectiveness of the acceleration factor.

From these results we observe the following points:

(i) The timing results decreases as the number of paths increases. This is correct up to a point close to the optimum value of $\omega$, then the timing results increase as the number of paths increases.
<table>
<thead>
<tr>
<th>Mesh size ((N \times N))</th>
<th>No. of paths</th>
<th>(\omega)</th>
<th>Time (seconds)</th>
<th>No. of iterations</th>
<th>No. of waiting cycles to access a critical section</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12x12)</td>
<td>1</td>
<td>1.0</td>
<td>148.590</td>
<td>162</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.3</td>
<td>81.800</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.4</td>
<td>65.280</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.5</td>
<td>48.750</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.6</td>
<td>31.300</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.7</td>
<td>38.810</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.8</td>
<td>55.330</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.9</td>
<td>134.760</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>90.730</td>
<td>189</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.3</td>
<td>56.370</td>
<td>117</td>
<td>285,48</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.4</td>
<td>50.050</td>
<td>102</td>
<td>50,95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.5</td>
<td>41.790</td>
<td>87</td>
<td>222,67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.6</td>
<td>38.160</td>
<td>78</td>
<td>17,65</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.7</td>
<td>41.270</td>
<td>84</td>
<td>22,75</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.8</td>
<td>156.020</td>
<td>320</td>
<td>146,255</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.9</td>
<td>1131.610</td>
<td>no convergence</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.0</td>
<td>67.930</td>
<td>199</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.3</td>
<td>43.500</td>
<td>131</td>
<td>1451,12,544</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.4</td>
<td>37.520</td>
<td>113</td>
<td>1204,13,474</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.5</td>
<td>33.470</td>
<td>97</td>
<td>1939,17,909</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.6</td>
<td>36.420</td>
<td>106</td>
<td>2111,26,986</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.7</td>
<td>336.360</td>
<td>no convergence</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>54.960</td>
<td>211</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.3</td>
<td>37.160</td>
<td>143</td>
<td>2264,11,1019,1429</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.4</td>
<td>32.770</td>
<td>126</td>
<td>1984,893,13,1250</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.5</td>
<td>28.900</td>
<td>111</td>
<td>1730,7,775,1153</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.6</td>
<td>250.740</td>
<td>959</td>
<td>17096,33,6582,10279</td>
</tr>
</tbody>
</table>

**TABLE 4.11:** The ASOR method to solve the two-dimensional Dirichlet problem
(ii) As a result from (i), the optimum $\omega$ decreases as the number of paths increases.

(iii) The range of convergence decreases when the number of parallel paths increases.

However, the exact optimum value of $\omega$ for any number of paths is obtained and these values are listed in Table 4.12 with the timing results and speed-up ratios. From Table 4.12, we see that no speed-up is achieved for this method when it is implemented on $k$ paths, $k=2,3,4$, so that the timing results are better when this algorithm was run sequentially, i.e. when $k=1$.

As the nature of the SOR method is known, the ASOR method is implemented with the critical sections to read and update the global data structure which holds the final results. It uses the new values of the components in the subset as soon as they are available within this subset and the update of the global data of all components within a set is performed within a critical section. Hence, when $k$ increases the dependent components will be in different subsets which are carried out by different processors running at different speeds. Therefore, the probability of obtaining the new values of the dependent components will be small which means that the components are forced to iterate on the old available values. This, however, can generate a significant delay that causes degradation in the problem's running-time. Therefore, no speed-up is achieved since the penalty paid for the critical section accesses is much more than the parallelism done in the algorithm.

To illustrate this degradation, it was considered worthwhile to examine the time spent to access the critical sections (C.S.) made by the algorithm and also the time that the processors spent on waiting for each other because the critical section is being used by another processor. However, the time required to access a critical section in the NEPTUNE system (i.e. the $\$ENTER/$\$EXIT
construct) is -400 μs seconds. Since in the implementation of the ASOR method there are critical sections, one is at the commencement of each path to read all components which is called just once, and the other critical section is to update the components of the subset of that path and to read in all the mesh components for the next cycle. The second critical section is called as many times as the path iterates on its components. Therefore, the number of accesses to critical sections is obtained by adding 1 to the number of iterations obtained experimentally. The corresponding timing required for the waiting cycles made by the process to enter a critical section is -200 μs seconds. Therefore, the time spent in the critical sections and for the processes to wait is calculated as follows:

\[ \text{Time spent in all c.s.} = \text{number of accesses to c.s.} \times 400 \mu s, \]

and

\[ \text{Time spent for waiting cycles to access c.s.} = \text{number of waiting cycles} \times 200 \mu s. \]

We calculate now the time spent in the critical sections for the ASOR experimental results and the overhead measurements of these critical sections accesses. In Table 4.13, such measurements are listed where by the static overhead we mean the actual accesses made by the algorithm which can be obtained from the output results, and the contention represents the time that the processes spend in waiting to access a critical section because it has been accessed by another process. The measurements in Table 4.13 are calculated from the results in Table 4.12.

The measurements in Table 4.13 declare that when the number of processes, \( k \), increases, which results in the increase of the number of iterations, the critical sections accesses also increase. Therefore, the contention amongst the processes increases which yields an unwanted delay that has reduced the value of the optimum \( \omega \).

It is also interesting to study the behaviour of the acceleration
<table>
<thead>
<tr>
<th>No. of paths</th>
<th>Optimum ( \omega )</th>
<th>Time (second)</th>
<th>No. of iterations</th>
<th>Speed-up</th>
<th>Waiting cycle to access critical section</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.62</td>
<td>25.880</td>
<td>28</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1.62</td>
<td>32.930</td>
<td>68</td>
<td>0.79</td>
<td>66</td>
</tr>
<tr>
<td>3</td>
<td>1.55</td>
<td>30.930</td>
<td>93</td>
<td>0.84</td>
<td>453</td>
</tr>
<tr>
<td>4</td>
<td>1.51</td>
<td>28.970</td>
<td>111</td>
<td>0.89</td>
<td>939</td>
</tr>
</tbody>
</table>

**TABLE 4.12:** The optimum \( \omega \) for the ASOR method

<table>
<thead>
<tr>
<th>No. of paths</th>
<th>Optimum ( \omega )</th>
<th>Time (second)</th>
<th>No. of iterations</th>
<th>Time to access c.s. (second)</th>
<th>c.s. access overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>static overhead</td>
<td>contention</td>
</tr>
<tr>
<td>1</td>
<td>1.62</td>
<td>25.880</td>
<td>28</td>
<td>0.012</td>
<td>0.04%</td>
</tr>
<tr>
<td>2</td>
<td>1.62</td>
<td>32.930</td>
<td>68</td>
<td>0.028</td>
<td>0.08%</td>
</tr>
<tr>
<td>3</td>
<td>1.55</td>
<td>30.930</td>
<td>93</td>
<td>0.038</td>
<td>0.12%</td>
</tr>
<tr>
<td>4</td>
<td>1.51</td>
<td>28.970</td>
<td>111</td>
<td>0.045</td>
<td>0.16%</td>
</tr>
</tbody>
</table>

**TABLE 4.13:** Critical sections overheads in the ASOR method
factor \( w \) for the Jacobi method. This means, a study of the JOR method which is given by equations (4.3.7) and (4.3.8). However, for our model problem, i.e. the two-dimensional Dirichlet problem, the Jacobi method cannot be accelerated by using \( w \) and this is because \( w=1 \) in this particular problem as it is shown below:

As the maximum and minimum eigenvalues of the Jacobi matrix \( B \) of equation (4.8.5) can be obtained, where the maximum and minimum eigenvalues of \( A \) are given in equation (4.8.4), thus,

\[
B = I - D^{-1}A,
\]

\[
\lambda_{\max} = 1 - \frac{h^2}{4}(\frac{8}{h^2} - \sin^2(\frac{h}{2}))
\]
\[
= -\cos h
\]

and

\[
\lambda_{\min} = 1 - \frac{h^2}{4}(\frac{8}{h^2} \sin^2(\frac{h}{2}))
\]
\[
= 1 - 2\sin^2 \frac{h}{2}
\]
\[
= \cos h
\]

This means that \( \lambda_{\max} = -\lambda_{\min} \), therefore, the optimum \( w \) becomes,

\[
\omega = \frac{2}{2-\cosh(-\cosh)} = 1
\]

This shows that the model problem used is not very suitable to illustrate the JOR method. Therefore, some experiments have been done on different types of matrices to implement the JOR method in order to investigate the behaviour of the under- or over-relaxation factor, \( \omega \), and the optimum \( \omega \) so that the system can converge as fast as possible.

The experiments have been performed on a matrix of coefficients of the linear system of equations which has the form,
where $A$ is of order $(n \times n)$ and $n$ was chosen as 16 and 32. The experimental results obtained from the NEPTUNE system are listed in Table 4.14, where (16 x 16) and (32 x 32) matrix sizes were used resulting in 16 and 32 equations to solve, respectively, with 4 processors cooperating in all cases. These results show that a linear speed-up was achieved. On the other hand, the optimum $\omega$ approximately matches the theoretical one especially when the number of paths is one or two.

The theoretical optimum $\omega$ for the case (16 x 16) matrix size is given as follows;

Since the maximum and the minimum eigenvalues of the matrix $A$ of order (16 x 16) are given respectively as 1.58187 and 0.955439, then by applying the formula (4.7.4), the optimum $\omega$ becomes 0.76.

From Table 4.14, we conclude that the optimum $\omega$ increases with the increase of the number of paths. The increase in the value of the optimum $\omega$ is evident in the case when 4 paths were co-operating, while in the case of 1 or 2 paths, the value of the optimum $\omega$ approximately equals the value given theoretically. On the other hand, in the case of implementing a matrix of order (32 x 32), the optimum $\omega$ was slightly different for any number of paths, 1, 2 and 4, and hence all of the values of the optimum $\omega$ are approximately equal to the theoretical value. Also, in this case, the
speed-up is more than the case of a \((16 \times 16)\) matrix and it is approximately linear. These results indicate that the JOR method seems more efficient for larger values of \(n\) for this type of matrix. The other important thing with this matrix is that the average iteration number decreases as the number of paths increases.

In these types of parallel asynchronous algorithms the estimation of the optimum \(\omega\) is difficult, (as in the SOR method) since they include critical sections to read and to update the global vector which holds the final correct answer to the problem. As indicated above, we cannot obtain the same \(\omega\) when a different number of paths are used. In fact, there are alternate regions of convergence and divergence as the number of parallel paths increases when \(\omega=0.7\) and \(0.8\). An explanation of the unstability that occurs for certain combinations of the number of paths and \(\omega\) is linked to the relative timing of the evaluation of the subset of components and the execution of the critical section in each path. As the number of paths increases, the number of the subsets increases, therefore, the components in each subset will be evaluated in a shorter time in each iteration step and afterwards, the parallel paths will attempt to access the critical section. This will force some paths to wait until the path currently executing the critical section terminates this part. It is clear that the JOR method is implemented as in the Jacobi method in which the components are all from a previous iteration which are updated in each subset after all of them are evaluated. This, however, allows the paths to communicate by means of critical sections to obtain the new values. Thus, the new values will be released and the next iteration step will make use of these new values. Accordingly, when the number of paths is small, the subset size is large, hence no update will occur in a short time and this causes most of the components of the next iterate to use the old values rather than the
new values. In general, we conclude that as the number of paths increases the probability of the divergence case to occur decreases when \( \omega \) increases. We also assert that the optimum \( \omega \) to guarantee the convergence test increases as well.

Moreover, as the size of the problem increases, the divergence case occurs in slightly different regions with the increase of the number of paths. This is because of the increase of the size of the problem which makes the subsets' size larger, hence the paths can spend their time doing useful computation before accessing the critical section. In fact, if we assume that the process spends more time in the evaluation section of the components than the critical section evaluation, then the multiprocessor system may be utilized when the size of the problem to be solved is large.

For comparison the SOR method is implemented to solve the same system of equations whose coefficient matrix \( A \) is given in (4.10.1). This method is implemented in a similar manner as the ASOR method which is used to solve the Dirichlet problem as discussed previously. The experimental results of this method are listed in Table 4.15 where (16×16) and (32×32) matrix sizes were used. The observations obtained from these results are similar to those obtained from the ASOR method above including the fact that no speed-up is achieved by this method and this is due to the reasons discussed in the ASOR method's implementation.

The two implementations of the SOR method lead to the same conclusion that because of the Gauss-Seidel method's basic iterations concepts together with the characteristics of the asynchronous application, the optimum \( \omega \) decreases significantly when the number of paths increases. This is different from the JOR method's implementation as presented by the experimental results above, and moreover, the following section will make it clear how the critical sections act in these two different methods.
<table>
<thead>
<tr>
<th>Matrix size</th>
<th>No. of paths</th>
<th>Optimum ( \omega )</th>
<th>Time (seconds)</th>
<th>No. of iterations</th>
<th>Speed-up</th>
<th>No. of waiting cycles for c.s.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(16\times16)</td>
<td>1</td>
<td>0.75</td>
<td>22.310</td>
<td>169</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.77</td>
<td>11.610</td>
<td>167</td>
<td>1.92</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.80</td>
<td>5.990</td>
<td>162</td>
<td>3.73</td>
<td>34</td>
</tr>
<tr>
<td>(32\times32)</td>
<td>1</td>
<td>0.75</td>
<td>162.020</td>
<td>601</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.76</td>
<td>82.240</td>
<td>582</td>
<td>1.97</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.77</td>
<td>40.890</td>
<td>550</td>
<td>3.96</td>
<td>229</td>
</tr>
</tbody>
</table>

**TABLE 4.14**: The JOR method solving the system of matrix \( A \)

<table>
<thead>
<tr>
<th>Matrix size</th>
<th>No. of paths</th>
<th>Optimum ( \omega )</th>
<th>Time (seconds)</th>
<th>No. of iterations</th>
<th>Speed-up</th>
<th>No. of waiting cycles for c.s.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(16\times16)</td>
<td>1</td>
<td>1.45</td>
<td>3.710</td>
<td>28</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.37</td>
<td>4.310</td>
<td>62</td>
<td>0.86</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.30</td>
<td>3.030</td>
<td>82</td>
<td>1.23</td>
<td>23</td>
</tr>
<tr>
<td>(32\times32)</td>
<td>1</td>
<td>1.67</td>
<td>14.760</td>
<td>55</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.61</td>
<td>16.730</td>
<td>121</td>
<td>0.88</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.43</td>
<td>13.550</td>
<td>182</td>
<td>1.09</td>
<td>48</td>
</tr>
</tbody>
</table>

**TABLE 4.15**: The SOR method solving the system of matrix \( A \)
4.11 THE BEHAVIOUR OF THE UNDER- AND OVER-RELAXATION ALGORITHMS

We have solved the Dirichlet problem involving the Laplace Equation on a grid of points using the five point difference formula with the SOR method involving an over-relaxation parameter $\omega$ which we described in more detail in the previous section. Another type of banded matrix of the form shown in formula (4.10.1) was used to investigate both the JOR method with an under-relaxation factor $\omega$ and the SOR method with an over-relaxation factor $\omega$.

Now, we consider the latter case in which the JOR and the SOR methods were implemented so that the update of the components of the subset of each path was protected by a critical section (C.S.) after which the values of the components become immediately available. As described before each path accesses the critical section as many times as it iterates on its subset, therefore the time required to access the critical sections is obtained by following the calculations made in the ASOR method. The results of these calculations together with the static overheads are presented in Table 4.16. The time required to access all critical sections in both methods is illustrated in Figure 4.14.

We notice that the number of iterations required to satisfy the convergence test decreased as the number of paths was increased in the JOR method, whereas in the SOR method, the number of iterations was significantly increased when the number of paths was increased. It follows that the number of accesses to the critical section increased accordingly.

Although the time required to access the critical sections in the case of the JOR method was greater than that of the SOR method, the critical section access overhead was the same in both cases for any given number of paths. However, Figure 4.14 illustrates that the required time to access the critical sections are slightly increased when the number of
FIGURE 4.13
The Timing Results of The PA and PAOR Iterative Methods

FIGURE 4.14
The Required Time to Access The Critical Section in The JOR and SOR Iterative Methods
paths increase, whereas the increase is significant in the SOR method. Therefore, we can see that the critical sections act as a bottleneck, especially in the SOR method, which tend to decrease the parallelism and increase the total running-time.

On the other hand, the increase of the number of iterations in the SOR method is presumably because of a redundant evaluation which occurs when the new values of the dependent components were not available since they have been evaluated by another path and updated by a critical section. This redundancy increases as more and more paths participate in the solution of the problem. Therefore, as the number of paths increases, the number of subsets increases which follows that the dependent components may be evaluated in different paths. This means that the evaluation of a component does not use the same ordering of the related components exactly as in the sequential algorithm. In addition, some of these orderings of the dependent components are critical, thus they may create instabilities sufficient to cause the iteration to diverge.

Rosenfeld and Driscoll [1969] and Rosenfeld [1969] presented some
discussion about what we presented above where they used a chaotic relaxation scheme to solve the Dirichlet problem. Also, their study includes a discussion about how the number of iterations was affected by the number of co-operating processors, some experimental results as well as some predicted results. However, most of their observations are similar to what we obtained in our particular implementations.

Now, for a precise discussion about the asynchronous algorithms in general and the algorithms with the acceleration factor in particular we proceed as follows:

Baudet [1978b] developed mathematical bounds for the rate of convergence and the efficiency of the synchronous Jacobi method. These bounds whose basic formulae are given in equations (4.5.6) and (4.5.7), are given for the Jacobi method where \( k \) paths are generated as

\[
R(k) > -\log \left( \frac{\rho(A)}{k} \right),
\]

which is the rate of convergence, and

\[
E_t(k) > -(k/\lambda_k) \log \rho(A),
\]

which is the efficiency measurement when the cost is measured by considering the average time to perform the evaluation. \( \rho(A) \) is, of course, the spectral radius of the original matrix \( A \). The factor \( \lambda_k \) is the penalty factor introduced by Kung [1976] which measures the overhead due to the fluctuations in the computing times of the \( k \) paths, and can be evaluated if we know, for example, the distribution function for the time to evaluate the function of the given system. In particular, \( \lambda_1 = 1 \) and for \( k \geq 2, \lambda_k > 1 \) with the equality sign valid when it takes the same constant time to evaluate the function (i.e. there is no fluctuation in the computing time).

The ratio \( E_t(k)/E_t(1) = \frac{k}{\lambda_k} \) measures the speed-up achieved by using a parallel implementation with \( k \) paths (processes). We would expect the implementation with \( k \) paths to be \( k \) times as efficient as the sequential
implementation (i.e. $k=1$), but this is not so because of the overheads introduced by synchronising the $k$ paths and measured by the penalty factor $\lambda_k$.

From the above-mentioned discussion, the average rate of convergence for the Jacobi method given in equation (4.11.1) depends on $k$, the number of paths. We can rewrite equation (4.11.1) so that the spectral radius of the Jacobi matrix is given as,

$$\rho(A) \leq e^{-Rk},$$

where $R(k)$ is written as $R$ for simplicity.

Therefore, if $k$ increases the spectral radius $\rho$ decreases and vice versa. It is known that when $R$ increases, faster convergence to the correct solution can be obtained, but slower convergence is obtained if $R$ decreases.

Now, if we consider the SOR iterative method, we need to know about the optimum value of the over-relaxation factor $\omega$. The optimum $\omega$ is obtained by the formula,

$$\omega_{\text{opt}} = \frac{2}{1 + \sqrt{4 - \rho^2(B)}}$$

where $\rho(B)$ is the spectral radius of the Jacobi iteration matrix. Since $\rho(B) \leq e^{-Rk}$, then if $R$ is held constant, $\rho(B)$ decreases as $k$ increases.

Therefore, when $k$ increases, $\rho(B)$ decreases which implies that $\omega_{\text{opt}}$ decreases. On the other hand, when $k$ decreases, $\rho(B)$ increases which follows that $\omega_{\text{opt}}$ increases as well. This means that the optimum $\omega$ for a given system of equations is decreased with the increase of the number of paths.

The above argument is adequate to explain our experimental results of the SOR method that solves the Dirichlet problem, see Table 4.13. From this table, we see that no speed-up was achieved for the optimum $\omega$ and this is related mainly to the existence of the critical sections where the static critical section access overheads in the case of the number of paths
equals 4 is larger than the case of only one path. Since four processors co-operate, therefore the contention is increased when 4 paths are generated while the contention is significantly small in the case when two paths were generated. Therefore, because of the critical sections in most of the asynchronous algorithms, the performance of such algorithms is degraded. The amount of degradation can be estimated analytically as follows:

Consider an asynchronous algorithm with k processes, \( p_1, \ldots, p_k \). Let \( T_i \) be the time taken by process \( p_i \) and \( c_i \) the time spent in the critical sections of the process \( p_i \). Define \( \alpha_i = c_i / T_i \) and let \( \alpha \) be a lower bound on the \( \alpha_1, \ldots, \alpha_k \). In general, an estimate on \( \alpha \) can be obtained by examining the programs of the processes. \( \alpha \) may or may not be a function of \( k \). In our iterative algorithms, as \( k \) increases the subsets of processes become smaller, therefore, the complexity of the subsets becomes smaller, i.e., \( T_i \) decreases. For this reason \( \alpha \) is an increasing function of \( k \), hence it can be written as \( \alpha(k) \). Since the execution of the critical sections in parallel algorithms cannot be overlapped, therefore, the speed-up factor of the algorithm is at most,

\[
\frac{T_1 + \ldots + T_k}{c_1 + \ldots + c_k} \leq \max \left( \frac{1}{\alpha_1}, \ldots, \frac{1}{\alpha_k} \right) = \frac{1}{\alpha(k)}.
\]

This indicates that the speed-up factor is also bounded by \( k \). Thus, an optimal choice of \( k \) exists, which is bounded above by the smallest positive solution of the equation \( k = \frac{1}{\alpha(k)} \).

The above arguments indicate that a large number of processes cannot help unless \( \alpha \) can be kept small. In particular, it is important to design algorithms which use small critical sections and to select the synchronisation tool which takes as little time as possible.
4.12 THE ANALYSIS OF THE ASYNCHRONOUS ITERATIVE ALGORITHMS

The analysis of many asynchronous algorithms is performed through the rules of probability theory as given by Robinson [1979] where he applied these rules mainly to sorting and merging algorithms. However, another approach for analysing asynchronous algorithms is by Baudet [1978a] who found that this analysis can be performed through a queueing model and order statistics concept.

Now, we follow Baudet's approach to analyse our asynchronous iterative algorithms. In this analysis two factors are to be considered:

1. The number of iterations required to solve the system of equations.
   This is obtained from the experiments because, estimating this number theoretically for the asynchronous algorithm is quite difficult. This is related to the fact that the behaviour of the asynchronous iterative methods is completely different to that of the sequential form.

2. The average time for each process (path) to execute a complete cycle i.e., from the instant it starts an evaluation to the instant it starts the next evaluation.

In the parallel implementation of asynchronous iterative algorithms, the processes cooperate in the evaluation of an iterate and have the form illustrated in Figure 4.15. The waiting section in the figure is only due to the presence of the critical section.

![Figure 4.15: The cyclic form of a process in the asynchronous iterative method](image)

FIGURE 4.15: The cyclic form of a process in the asynchronous iterative method
Now, let us assume that the execution times for the evaluation section (ES) by all k processes are independent identically distributed random variables distributed according to the probability distribution $F_k$, associated with the density function $f_k$. Let $\tau_k$ and $\sigma_k$ denote their mean and variance, respectively. Similarly, we assume that the execution times for the critical section by all k processes are independent identically distributed random variables distributed according to the probability distribution $G_k$, associated with the density function $g_k$. Let $c_k$ denote their mean. Let the parameter for $F_k$ be $\lambda = \frac{1}{\tau_k}$ and for $G_k$ be $\mu = \frac{1}{c_k}$, when $F_k$ and $G_k$ are exponentially distributed.

Now, we can represent the parallel implementation of the asynchronous algorithm with k processes carried out by P processors when serviced by the queueing system depicted in Figure 4.16.

Figure 4.16 has the following properties:

(a) k processes in the whole system
(b) P servers in system (1): the evaluation section
(c) 1 server in system (2): the critical section
(d) with the restriction that at most P servers are active at the same time in both systems (1) and (2).
Let us assume that \( q_i \) be the steady state probability (i.e., when the probability is not changed with the time, the system is said to have reached a steady state and the behaviour of the queue over a long period of time will be dominated by the steady state probability) and that \( i \) customers are in system (1) of Figure 4.16. This implies that \( i \) processes are executing their evaluation sections, while \((k-i)\) processes are ready to execute the critical section, where \( i=0,1,...,k \). Let \( \tau_0 \) denote the probability that no process is executing the critical section, either because all the processes are within their evaluation sections or because no processor is allocated to a process ready to execute the critical section. Therefore, if we assume that there exists at any time in the entire system \( i \) processes with \( i>P \), which are not blocked (waiting for another process to complete the critical section), each of the \( i \) processes receives the same fraction \( \frac{P_i}{i} \) of the computing power. Hence, it follows that the probability \( \tau_0 \) is given by

\[
\tau_0 = q_k + \sum_{p \leq i \leq k-1} \frac{i-P+1}{i+1} q_i.
\]

The term \( q_k \) in the RHS of equation (4.12.1) corresponds to the probability that all the processes are within their evaluation section. The second term corresponds to the probability that some processes are blocked (i.e., waiting to access the critical section because no processor is available to allocate them). To explain this term further, assume that \( P=4 \), hence if \( i=P=4 \) the entire system will hold 5 processes (4 in the ES and 1 in the CS). Since only 4 processors are available at any time, therefore, one process will be forced to wait for a processor and the probability of this is \( \frac{1}{5} \). When \( i=5 \), this means that 6 processes are in the whole system, 4 of them are allocated to processors and 2 are waiting, hence \( \frac{2}{6} \) is the probability of the waiting processes. This probability is obtained from \( (1 - \frac{P}{i+1}) = \frac{i-P+1}{i+1} \). Since \( \frac{P}{i+1} \)
is the computing power each process can receive, therefore, \(1 - \frac{P}{i+1}\) is the probability of not receiving any computing power when \((i+1)\) processes are in the entire system.

Baudet [1978b] has developed the average time required to evaluate a complete cycle of the iterative asynchronous algorithm which is presented by the following theorem.

**Theorem 4.12.1**

Assume that \(k \geq P\). The average time \(t_k\) that is required to execute a complete cycle is given by,

\[
t_k = \frac{k \cdot c_k}{(1 - \pi_0)} + \tau_k, \quad (4.12.2)
\]

where \(\pi_0\) is the probability that the server of system (2) is idle (i.e., no process is executing the critical section, although some may be blocked because no processor is available). If we assume that each process which is not blocked receives an equal share of the computing power, the probabilities \(q_i\), for \(i=0,1,\ldots,k\), will be as follows:

\[
q_i = \begin{cases} 
q_k & \text{if } i=k, \\
(i+1) \frac{(k-1)!}{i!} \beta^{(k-i)} q_k & \text{if } P \leq i < k-1, \\
p \frac{(k-1)!}{i!} \beta^{(k-i)} q_k & \text{if } 0 \leq i < P-1.
\end{cases} \quad (4.12.3)
\]

It is required to present some concepts from queueing theory to provide clarity for the proof of the above theorem.

**Note (1)**

Let \(\lambda\) be the average arrival rate of customers to the queueing system, and \(\mu\) be the average service rate. Define \(\rho\) as \(\frac{\lambda}{\mu}\) for a single server and \(\rho = \frac{\lambda}{\mu m}\) for \(m\)-servers. Thus,

\[\rho = \mathbb{E}[\text{fraction of busy servers}],\]

which is the expectation of the fraction of the busy servers. Indeed, in order for a system with a general distributed function and one-server to be stable, \(\rho\) must be in the range \(0 < \rho < 1\). Stability here refers to the fact that
limiting distributions for all random variables served. In such cases, we come to the following calculation.

Let $\tau$ be an arbitrarily long time interval; during this interval we expect with probability 1 that the number of arrivals will be very nearly equal to $\lambda \tau$. Moreover, let $P_0$ denote the probability that the server is idle at some selected time. We may, therefore, say that during the interval $\tau$, the server is busy for $(\tau - \tau P_0)$ seconds, and so with probability 1, the number of customers served during the interval $\tau$ is nearly $\mu(\tau - \tau P_0)$. Now, we equate the number of arrivals to the number served during this interval, which gives for large $\tau$,

$$\lambda \tau \approx \mu(\tau - \tau P_0).$$

Thus, as $\tau \to \infty$, we have $\frac{1}{\mu} = 1 - P_0$; using the definition of $\rho$ above we finally obtain the important conclusion for a general distributed function system,

$$\rho = 1 - P_0 \quad \text{(Kleinrock [1975], pp.19)}$$

That is, $\rho$ is merely the fraction of time the server is busy.

**Note (2)**

Little's result (Kleinrock [1975], pp.17) is stated that the average number of customers in a queueing system is equal to the average arrival rate of customers to that system, times the average time spent in that system. In notation,

$$\bar{N} = \lambda T,$$

where $\bar{N}$ is the average number of customers in the system, $\lambda$ is the average arrival rate, and $T$ is the average time spent in that system. However, the above result does not depend on the arrival or the service probability distributions neither does it depend on the number of servers in the system.

**Note (3)** (Kleinrock, 1975, pp.22 and Ch.3)

The "birth-death process" is a very important special class of Markov chains. This process may be either a discrete-or continuous-time process in
which the defining condition is that state transitions take place between neighbouring states only. Thus, if we choose the set of integers as the discrete state space, then the birth-death process requires that if \( x_n = i \), then \( x_{n+1} = i-1, i \) or \( i+1 \) and no other. However, birth-death processes have played a significant role in the development of queueing theory. Usually, the continuous-time birth-death process is of most interest in which transitions only to neighbouring states \( E_{k+1} \) or \( E_{k-1} \) from state \( E_k \) are permitted. Moreover, \( E_k \) represents the state of the system when the population size is \( k \) at that time. The transition from \( E_k \) to \( E_{k+1} \) signifies a "birth" in the population, whereas a transition from \( E_k \) to \( E_{k-1} \) denotes a "death" in the population. Therefore, we define \( \lambda_k \) as a "birth-rate" at which births occur when the population size is \( k \). Similarly, we define a "death-rate" \( \mu_k \) at which deaths occur when the population size is \( k \). Since these birth and death rates are independent of time and depend only on \( E_k \), therefore, we have a continuous-time homogeneous Markov chain of the birth-death type. If we let, 

\[
\lambda_k = P_{k,k+1}, \text{ i.e. transition from state } k \text{ to } k+1,
\]

and 

\[
\mu_k = P_{k,k-1}, \text{ i.e. transition from state } k \text{ to } k-1,
\]

and the nearest-neighbour condition requires that \( P_{kj} = 0 \) for \( |k-j| > 1 \). Since \( \sum_j P_{kj} = 0 \) (Kleinrock [1975], pp.49), then we require \( P_{kk} = -(\mu_k + \lambda_k) \) to satisfy the condition \( \sum_j P_{kj} = 0 \). Therefore, the infinitesimal generator for the general homogeneous birth-death process takes the form,

\[
P = \begin{bmatrix}
-\lambda_0 & \lambda_0 & 0 \\
\mu_1 & - (\lambda_1 + \mu_1) & \lambda_1 \\
\mu_2 & - (\lambda_2 + \mu_2) & \lambda_2 \\
\mu_3 & - (\lambda_3 + \mu_3) & \lambda_3 \\
0 & \cdots & \cdots 
\end{bmatrix}
\]

The \( P \) matrix is a tridiagonal matrix, all entries of which are zero
except for the main, upper and lower diagonals. The matrix $P$ is called the state-transition rate matrix whose information can be displayed by means of the state-transition rate diagram. In such a diagram, the state $E_k$ is represented by an oval surrounding the number $k$. Each non-zero infinitesimal rate $P_{ij}$ (an element of the matrix $P$) is represented in the state-transition rate diagram by a directed branch pointing from $E_i$ to $E_j$ and labelled with the value $P_{ij}$. Furthermore, since it is clear that the terms along the main diagonal of $P$ contain no new information we do not include the "self-loop" from $E_i$ back to $E_i$. Thus, the state-transition rate diagram for the general birth-death process is as shown in Figure 4.17.

![Figure 4.17: The state-transition rate diagram for the birth-death process](image)

Now, we define $q_i$ to be the limiting stationary distribution and the probability vector $Q$ as $Q=[q_0, q_1, \ldots]$. To find the probabilities $q_i$, we need to solve the matrix equation,

$$QP = 0.$$  \hfill (4.12.4)

This equation coupled with the probability conservation relation, namely

$$\sum_i q_i = 1,$$

uniquely gives us our limiting state probabilities. However, equation (4.12.4) is important for the continuous-time Markov chain and it is different from the discrete-time Markov chain equation which is written as,

$$QP' = 0.$$  

Here, $P'$ is the matrix of transition probabilities, whereas the infinitesimal generator $P$ is a matrix of transition rates.
A simple continuous-time Markov process is used to represent the operation of a single-process, time-shared system. The states of the Markov process representing a system with \( n \) users will correspond to the number of users in the working part of the interaction. That is, state \( j \) being occupied will indicate that \( j \) users are in the working portion of an interaction. Thus, the Markov process will have \( (n+1) \) states. In order to use the continuous-time Markov process, the distributions of processor-time per interaction and the duration of the console portion of the iteration must be exponential. The mean time for the console portion will be \( T \); the mean processor time per interaction is \( \frac{1}{P} \). Now if there are currently \( j \) users waiting for service (i.e., in the working part of an interaction), the rate to exit to the state where \((j+1)\) users are waiting for service is \( \frac{(n-j)}{T} \); in other words, \((n-j)\) users are in the console portion of the interaction. The rate of exit to the state where there are \((j-1)\) users is \( \frac{1}{P} \). This implies that each of the \( j \) users is receiving \( \frac{1}{j} \) of the processors' capacity. Thus, each user finishes at a rate of \( \frac{1}{jP} \) and any one of the \( j \) users finishes at a rate of \( \frac{1}{P} \). Matrix \( A \) in form (4.12.5) shows the transition-rate matrix,

\[
A = \begin{bmatrix}
-\frac{n}{T} & \frac{n}{T} & & & \\
\frac{1}{P} & -(\frac{1}{P} + \frac{n-1}{T}) & \frac{n-1}{T} & & \\
\frac{1}{P} & -(\frac{1}{P} + \frac{n-2}{T}) & \frac{n-2}{T} & & \\
\frac{1}{P} & -(\frac{1}{P} + \frac{n-3}{T}) & \frac{n-3}{T} & & \\
 & & & & 0
\end{bmatrix}
\]

(4.12.5)
An expression for the steady-state probabilities is found as follows: Let 
\( \pi = (\pi_0, \pi_1, \ldots, \pi_n) \) be the vector of the steady-state probabilities of occupancy.

Then, \( \pi A = 0 \).

The equations for the \( \pi_i \)'s are:
\[
\begin{align*}
-\frac{n}{T} \pi_0 + \frac{1}{P} \pi_1 & = 0, \\
\frac{n}{T} \pi_0 - \frac{1}{P} \pi_1 - \frac{n-1}{T} \pi_1 + \frac{1}{P} \pi_2 & = 0.
\end{align*}
\]

By solving the above two equations we obtain:
\[
\begin{align*}
\pi_1 & = n \frac{P}{T} \pi_0, \\
\pi_2 & = n(n-1) \left( \frac{P}{T} \right)^2 \pi_0.
\end{align*}
\]

In general,
\[
\pi_i = \frac{n!}{(n-i)!} \left( \frac{P}{T} \right)^i \pi_0.
\]

Making use of the fact that \( \sum_{i=0}^{n} \pi_i = 1 \), the following equation results:
\[
1 = \pi_0 \left[ 1 + n \frac{P}{T} + n(n-1) \left( \frac{P}{T} \right)^2 + \ldots + \frac{n!}{(n-j)!} \left( \frac{P}{T} \right)^j + \ldots + n! \left( \frac{P}{T} \right)^n \right].
\]

Let \( \frac{P}{T} = \tau \) and solve for \( \pi_0 \) to yield,
\[
\pi_0 = \frac{1}{\sum_{j=0}^{n} \frac{n!}{(n-j)!} \tau^j}.
\]

Thus,
\[
\pi_i = \frac{n!}{(n-i)!} \frac{\tau^i}{\sum_{j=0}^{n} \frac{n!}{(n-j)!} \tau^j}.
\]

\( \pi_0 \) is the steady-state probability that the processor is idle.

Now, a model for the operation of an \( m \) processors, \( n \) users time-shared system can be derived. From a state where \( j \) users are waiting for service, the exit rate to the state where there are \( (j+1) \) users waiting for service is \( \frac{(n-j)}{T} \). If \( j \) is less than \( m \), then each user's program is assigned its own processor and the rate of exit to the state where \( (j-1) \) users are waiting for service is \( \frac{1}{P} \). If \( j \) is greater than or equal to \( m \), the \( j \) users share the \( m \) processors just as in the case of a single processor, and the rate of
exit to the state where \((j-1)\) users are waiting is \(\frac{m}{p}\). The rate matrix of this process is shown in (4.12.6) as matrix \(B\).

\[
B = \begin{pmatrix}
-\frac{n}{T} & \frac{n}{T} \\
\frac{1}{p} & \frac{1}{p} + \frac{n-1}{T} & \frac{n-1}{T} \\
\frac{2}{p} & \frac{2}{p} + \frac{n-2}{T} & \frac{n-2}{T} & \ddots \\
\frac{3}{p} & \frac{3}{p} + \frac{n-3}{T} & \frac{n-3}{T} & \ddots & \ddots \\
\vdots & \vdots & \vdots & \ddots & \ddots & \ddots \\
\frac{m}{p} & \frac{m}{p} + \frac{n-m}{T} & \frac{n-m}{T} & \ddots & \ddots & \ddots & \ddots \\
0 & 0 & 0 & \ddots & \ddots & \ddots & \ddots & \ddots \\
\end{pmatrix}
\]

By solving the equation \(\tau B = 0\), we obtain,

\[
\tau_0 = \left\{ \sum_{i=0}^{n} \frac{n!}{i!(n-i)!} x_i \right\}^{-1},
\]

\[
\tau_i = \frac{n!}{i!(n-i)!} \frac{x_i}{\sum_{j=m}^{m} \frac{n!}{j!(n-j)!} x_j} \quad \text{for } i \geq m,
\]

and for \(i \geq m\),

\[
\tau_i = \frac{n!}{m!(n-i)!} \frac{x_i}{\sum_{j=m}^{m} \frac{n!}{j!(n-j)!} x_j} \quad \text{for } i \geq m.
\]

Now, we can present the proof of the Theorem (4.12.1) as stated above.

**Proof:**

Equation (4.12.2) in the theorem follows directly from Little's result.
as discussed in Note (2) above. That is, the average time $t_k$ spent in the entire system can be calculated as follows:

Since we have $\frac{\lambda}{\mu} = \rho$ (Note (1)), then,

$$\lambda = \mu \rho = \mu (1 - p_0) \quad \text{(Note (1))}.$$

From Little's result and if we consider the throughput of system (2) of Figure 4.16, we have,

$$t'_k = \frac{k}{\lambda} = \frac{k}{\mu (1 - p_0)},$$

where $\lambda$ is the average arrival rate. But in our queueing model $\frac{1}{\mu}$ represents the mean $c_k$ of the random variables of the execution times for the critical section of all $k$ processes. Now, since $\pi_0$ is the probability that the server in system (2), Figure 4.16, is idle then $(1 - \pi_0)$ is the probability that the server in system (2) is busy. Thus,

$$t'_k = \frac{k c_k}{(1 - \pi_0)}.$$

Here, $t'_k$ represents the average time spent in system (2) for one cycle of the iterative algorithm.

For a measure of the total time of the iterative algorithm, we have,

$$T_k = t'_k N_k,$$

where $t'_k = t'_k + t''_k$, $N_k$ represents the number of iterations that are required to solve the system of equations. $N_k$ is a function of $k$ and it is quite difficult to obtain this quantity theoretically. Therefore, it is obtained from the experimental results.

Equation (4.12.3) of Theorem (4.12.1) can be obtained directly from the fact that (under the exponential assumption for both $F_k$ and $G_k$) the system of Figure 4.16 corresponds to a pure birth-death process.

To obtain the $q_1$ of equation (4.12.3), we consider Note (3) and (4) to derive our transition-rate matrix. Our queueing model contains $k$ processes in the entire system (system (1) and (2)) of Figure 4.16. We assume that $p_k$ processors co-operate to carry out these processes. Since,
is the steady-state probability that \( i \) processes are in system (1) and the remainder, \((k-1)\) processes are ready to evaluate the critical section, therefore, if we consider that system (1) has \( k \) processes at the beginning of the algorithm, then the system will transit from state \( k \) to state \((k-1)\) where \((k-1)\) processes are in system (1) and 1 process is ready to enter system (2).

We assume that the mean processor time per process evaluation is \( \beta \).

Now, if there are \( j \) processes in system (1) with \( p < j < k \), then the rate of exit to the state where \((j-1)\) processes are in system (1) is \( \frac{j-1}{j} \). The rate of exit to the state where there are \((j+1)\) processes is \( \frac{1}{j\beta} \) which implies that each of the \( j \) processes is receiving \( \frac{1}{j} \) th of the processors capacity. Thus, each process can finish at a rate of \( \frac{1}{j\beta} \). If \( 0 < j < p \), the rate of exit to the state where \((j+1)\) processes are in system (1) is \( \frac{1}{p\beta} \) and the rate of exit to the state where \((j-1)\) processes are in system (1) is \( \frac{j-1}{p} \).

In the above discussion, we assumed that, corresponding to our queueing model, the "birth" to a new state occurs through the transition from state \( j \) to state \((j-1)\) and the "death" occurs when the transition is from state \( j \) to state \((j+1)\). This is because we started with \( k \) processes that are evaluated concurrently and they join the queue when they try to enter the critical section. In other words, to transit from state \( k \) to state \((k-1)\), means that 1 process will be ready for system (2). To transit from \((k-1)\) to \((k-2)\) means that now, 2 processes are ready for system (2) and so on. Hence, we can draw the transition-rate matrix for this queueing model.
Let our probability vector be \( Q = [q_k, q_{k-1}, \ldots, q_2, q_1, q_0] \). Then, we solve the equation, 

\[ QA = 0 . \]

Thus,

\[ a_k = \frac{1}{k\beta} q_{k-1} \]

\[ a_{k-1} = k\beta q_k . \]

Since

\[ a_k - \frac{1}{k\beta} q_{k-1} - \frac{k-1}{k} q_{k-1} + \frac{1}{(k-1)\beta} q_{k-2} = 0 \]

\[ a_{k-2} = (k-1)\beta \cdot \frac{k-1}{k} \cdot k\beta q_k = \beta^2 (k-1)(k-1) q_k . \]

Similarly,

\[ a_{k-3} = (k-2)\beta \cdot \frac{k-2}{k-1} \beta^2 (k-1)(k-1) q_k \]

and

\[ a_{p-1} = \beta (k-p+1) \cdot p(k-1)(k-2) \ldots (k-p) q_k . \]

Therefore,

\[ a_1 = (i+1) \cdot \frac{(k-1)!}{1!} \beta (k-1) q_k , \text{ for } p \leq k-1. \]
\[ q_i = p \cdot \frac{(k-1)!}{i!} \beta^{(k-i)} q_k, \text{ for } 0 \leq i \leq p-1 \]

\[ q_i = q_k, \text{ for } i = k. \]

Hence, the proof of Theorem (4.12.1) is complete.

Now, we can calculate \( q_k \) from the fact that \( \sum_{i=0}^{k} q_i = 1 \). Thus,

\[ q_0 + q_1 + \ldots + q_k = 1 \]

\[ q_k \cdot \left[ 1 + \sum_{i=0}^{p-1} p \cdot \frac{(k-1)!}{i!} \beta^{(k-i)} + \sum_{i=p}^{k-1} (i+1) \cdot \frac{(k-1)!}{i!} \beta^{(k-i)} \right] = 1 \]

\[ q_k = \left[ 1 + \sum_{i=0}^{p-1} p \cdot \frac{(k-1)!}{i!} \beta^{(k-i)} + \sum_{i=p}^{k-1} (i+1) \cdot \frac{(k-1)!}{i!} \beta^{(k-i)} \right]^{-1} \]

Therefore, for \( 0 \leq i \leq p-1, \)

\[ q_i = p \cdot \frac{(k-1)!}{i!} \beta^{(k-i)} \cdot \left[ \sum_{j=0}^{p-1} p \cdot \frac{(k-1)!}{j!} \beta^{k-j} + \sum_{j=p}^{k-1} ((j+1) \cdot \frac{(k-1)!}{j!} \beta^{k-j}) \right] \]

and for \( k-1 \leq i \leq p, \) we have,

\[ q_i = (i+1) \cdot \frac{(k-1)!}{i!} \beta^{(k-i)} \cdot \left[ \sum_{j=0}^{p-1} p \cdot \frac{(k-1)!}{j!} \beta^{k-j} + \sum_{j=p}^{k-1} ((j+1) \cdot \frac{(k-1)!}{j!} \beta^{k-j}) \right]^{-1} \]

Since all \( q_i \)'s are known now, then we can solve for \( \pi_0 \) and \( t_k \). First, assume that the expression for \( q_k = y \), then,

\[ \pi_0 = q_k + \sum_{i=p}^{k-1} q_i \]

\[ = y + \frac{1}{p+1} q_p + \frac{2}{p+2} q_{p+1} + \ldots + \frac{k-p}{k} q_{k-1} \]

\[ = y + \frac{1}{p+1} \cdot (p+1) \cdot \frac{(k-1)!}{p!} \beta^{k-p} \cdot y + \frac{2}{p+2} \cdot (p+2) \cdot \frac{(k-1)!}{(p+1)!} \beta^{(k-p+1)} \cdot y \]

\[ + \ldots + \frac{k-p}{k} \cdot \frac{(k-1)!}{(k-1)!} \beta^{k-1} \cdot y \]

\[ = y \left[ 1 + \frac{(k-1)!}{p!} \beta^{k-p} + \frac{2(k-1)!}{(p+1)!} \beta^{k-p+1} + \ldots + \frac{(k-p)(k-1)!}{(k-1)!} \beta \right] \]

\[ = y \left[ 1 + \sum_{i=p}^{k-1} (i+1) \frac{(k-1)!}{i!} \beta^{k-i} \right] \]
The formula for the time spent in one full cycle of the iterative method is then given by,

\[ t_k = \frac{k \cdot c_k}{1 - \pi_0} + \tau_k, \]

where, \( c_k \) is the mean time which is spent in the critical section of the cycle, \( k \) is the number of processes, and \( \pi_0 \) is the probability that no process is executing the critical section, therefore \( 1 - \pi_0 \) is the probability that the critical section server is busy. Here, we assumed that \( \tau_k \) is the mean time for the evaluation section.

In case of finding the timing results for all asynchronous algorithms theoretically, we find \( c_k \) as follows:

Time for the first do loop in the critical section (see for example program A.3) which includes the do increments and the update for the variable plus the time for the second loop to read all components for the mesh. The value of \( \tau_k \) is the complexity for each component multiplied by the number of the components in the subset of the process.

Thus, for a (12x12) mesh size the Asynchronous Gauss-Seidel algorithm requires the following time in the critical section:

\[ c_k \text{ values:} \]

- **update components**
- **read components**

<table>
<thead>
<tr>
<th>( k = 1 )</th>
<th>( c_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do increment</td>
<td>( 12\mu s \times 144 + 12\mu s \times 144 + 12\mu s \times 196 + 12\mu s \times 196 = 8160\mu s )</td>
</tr>
<tr>
<td>Do incr. copy opr.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( k = 2 )</th>
<th>( c_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do increment</td>
<td>( 12\mu s \times 72 + 12\mu s \times 72 + 12\mu s \times 196 + 12\mu s \times 196 = 6432\mu s )</td>
</tr>
<tr>
<td>Do incr. copy opr.</td>
<td></td>
</tr>
</tbody>
</table>
where, the required time for a Do loop increment is 12 µs on the NEPTUNE system, and a copy operation requires 12 µs (Barlow et al [1981]).

Similarly, to measure the time spent in the evaluation section in each process, we need to know the required time to perform an arithmetic operation on the NEPTUNE system which is 700 µs. Therefore, the \( \tau_k \) values are,

when \( k=1 \),
\[
\tau_k = 7 \text{ flops} \times 700 \, \mu s \times 144 + 12 \, \mu s \times 144 = 707328 \, \mu s
\]
when \( k=2 \),
\[
\tau_k = 7 \text{ flops} \times 700 \, \mu s \times 72 + 12 \, \mu s \times 72 = 353664 \, \mu s
\]
where flops denotes floating point operations.

It was mentioned that the total time \( T_k = N_k \cdot \tau_k \), where \( N_k \) is the number of iterations that is obtained from the experiments. Therefore, we can list the theoretical results of the \( c_k \), \( t_k \), and \( T_k \) for the AGS, AJ and ASOR methods which are discussed above to solve the 2-dimensional Dirichlet problem. Also, we list the results for the AJOR and ASOR methods that solve the special problem of matrix A in the form (4.10.1) in Section (4.10). For the latter two methods we use a \((32 \times 32)\) matrix size, where 14 flops are performed, and we denote the two methods for clarity AJOR and ASOR1. The theoretical results of these five methods are listed in Table 4.17. The theoretical and experimental results are illustrated diagrammatically in Figure 4.18.

It is clear that the theory of asynchronous iterative algorithms fairly matches the experimental results obtained from the NEPTUNE system. In our experiments, \( k=p \) and still reasonable agreement with the theory is obtained. However, in the AJ and AGS methods, the experimental results are slightly greater than those obtained from the theory, whereas the results are opposite in the case of the AJOR, ASOR and ASOR1 methods (see Figure 4.18).

Another observation can be obtained from Figure 4.18 and that is in the SOR methods, the theoretical timing results in case \( k=2 \) were greater than that for case \( k=1 \) and this confirms what was obtained from the experimental results.
This, however, is due to the fluctuation of time within a process which acts asynchronously. Although, the AJOR method implements critical sections, the experimental results match the theory and both show a gain of speed-up when more than one process is associated with the problem. From this case, we conclude that the number of iterations also affects the timing of the problem. Since the number of iterations decreases when $k$ increases in the AJOR method while the increase was significant in the ASOR1. This increase is obtained from the natural behaviour of the method.

To summarize, the theory for the asynchronous iterative methods presented here was adequate to verify the experiments carried out on the NEPTUNE system and a good comparison between the two results was obtained.
<table>
<thead>
<tr>
<th>Method</th>
<th>k</th>
<th>$(1-\pi_0^{-1})$</th>
<th>$c_k$ (µs)</th>
<th>$\tau_k$ (µs)</th>
<th>$\tau_k$ (µs)</th>
<th>$N_k$</th>
<th>Total time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGS (12x12) mesh</td>
<td>1</td>
<td>1.09375</td>
<td>8160</td>
<td>707328</td>
<td>716253</td>
<td>162</td>
<td>Theory: 116.033</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.09375</td>
<td>6432</td>
<td>353664</td>
<td>367734</td>
<td>189</td>
<td>69.502</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.046875</td>
<td>5856</td>
<td>235776</td>
<td>254167</td>
<td>199</td>
<td>50.580</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.015625</td>
<td>5568</td>
<td>176832</td>
<td>199452</td>
<td>211</td>
<td>42.10</td>
</tr>
<tr>
<td>AJ (12x12) mesh</td>
<td>1</td>
<td>same as in AGS</td>
<td></td>
<td></td>
<td></td>
<td>306</td>
<td>219.173</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>&quot;</td>
<td></td>
<td></td>
<td></td>
<td>321</td>
<td>118.043</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>&quot;</td>
<td></td>
<td></td>
<td></td>
<td>326</td>
<td>82.860</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>&quot;</td>
<td></td>
<td></td>
<td></td>
<td>333</td>
<td>66.420</td>
</tr>
<tr>
<td>ASOR (12x12) mesh</td>
<td>1</td>
<td>1.09375</td>
<td>8160</td>
<td>1009728</td>
<td>1018653</td>
<td>28</td>
<td>28.520</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.09375</td>
<td>6432</td>
<td>504864</td>
<td>518934</td>
<td>68</td>
<td>35.289</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.046875</td>
<td>5856</td>
<td>336576</td>
<td>3549675</td>
<td>93</td>
<td>33.012</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.015625</td>
<td>5568</td>
<td>252432</td>
<td>275052</td>
<td>111</td>
<td>30.530</td>
</tr>
<tr>
<td>AJOR (32x32) matrix</td>
<td>1</td>
<td>1.09375</td>
<td>1584</td>
<td>313984</td>
<td>315717</td>
<td>601</td>
<td>189.745</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.09375</td>
<td>1200</td>
<td>156992</td>
<td>158304</td>
<td>582</td>
<td>92.133</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.015625</td>
<td>1088</td>
<td>78496</td>
<td>79520</td>
<td>550</td>
<td>43.736</td>
</tr>
<tr>
<td>ASOR1 (32x32) matrix</td>
<td>1</td>
<td>same as in AJOR</td>
<td></td>
<td></td>
<td></td>
<td>55</td>
<td>17.364</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>&quot;</td>
<td></td>
<td></td>
<td></td>
<td>121</td>
<td>19.155</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>&quot;</td>
<td></td>
<td></td>
<td></td>
<td>182</td>
<td>14.473</td>
</tr>
</tbody>
</table>

**TABLE 4.17:** Theoretical and Experimental results for the different implementations of the asynchronous algorithms.
FIGURE 4.18
The Theoretical and Experimental Values of The Asynchronous Iterative Methods
4.13 CONCLUSIONS

This chapter included the study of asynchronous iterative methods where the problem to be solved was partitioned into $k$ processes with $k \geq p$, the number of the available processors. The processes were not synchronised and each process iterates on its subset using the new values of the components of the mesh when they are available. Hence, the processes have a minimum of intercommunication through critical sections to allow processes to exchange their results to complete the solution of the problem. The Purely Asynchronous method described in Section 4.9 is a typical asynchronous method with no critical sections at all. However, all asynchronous iterative methods show a great improvement over the synchronous methods that synchronise the processes at the end of each iteration step, which in turn, degrades the performance of the algorithms.

Many experiments have been performed on the NEPTUNE asynchronous system of 4 processors, and several asynchronous iterative methods have been implemented to solve a large system of equations drawn from the finite difference approximations, in particular, the two-dimensional and the one-dimensional Dirichlet problems. These methods use the Jacobi method with full synchronisation, the AJ and AGS methods with critical sections and the PA method with no synchronisation at all. However, the experimental results show a good comparison between these methods with the PA method being the best among them because it exhibits full parallelism and has a linear speed-up. The Jacobi method is considered the worst as it implements synchronisation.

Two strategies of decomposing the problem were used, the sequential and non-sequential allocation of the rows of the mesh to each process. It was observed, from the experimental results, that the sequential decomposition gave better results so that it was considered in all of the other applications completed in this chapter.
The other factor obtained from the experiments of the two-dimensional problem is the number of iterations which remains the same for any number of processes in the Jacobi and PA methods, and linearly increased with the number of processes in the case of the AJ and AGS methods. Whilst in the one-dimensional problem, the number of iterations decreases as the number of processes increases for the Jacobi, AJ and PA methods, whereas the iterations increase when the number of processes increases in the case of the AGS method.

Some experiments were also performed on another asynchronous iterative method with an acceleration factor $\omega$. The Purely Asynchronous Over-Relaxation method (PAOR) showed a great improvement over the PA method and it has a linear speed-up as well. Whereas, the AGS method with $\omega$, i.e. the ASOR method did not gain any speed-up. Moreover, the optimum value of $\omega$ was decreased with the increase of the co-operating processes. This was shown to be related to the instability of the spectral radius of the matrix of the problem when the number of processes increases and this however is due to the existence of critical sections.

On the other hand, the JOR and the SOR methods were implemented asynchronously to solve a special type of matrix which was given in equation (4.10.1). With the JOR method, a linear speed-up was achieved when 32 equations were solved. Also, the optimum value of $\omega$ was almost the same in any number of processes, and the number of iterations decreased as the number of processes increased. Whereas, the results of the SOR method has a similar behaviour to those of the Dirichlet problem. Actually, these two methods are implemented in the same manner but the time spent in the evaluation of the critical sections was increased as the number of processes increases in the SOR method, and vice versa in the JOR method.
CHAPTER 5

PARALLEL IMPLEMENTATIONS OF THE FOUR-POINT BLOCK EXPLICIT METHOD
5.1 INTRODUCTION

The iterative methods studied in Chapter 4 were solved considering each point of the grid mesh individually. A faster rate of convergence may be obtained when a group of points is formed in a way that all the points in one group are evaluated at once in one iteration step rather than solving the individual points. An example of such a method is the Line Successive Over-Relaxation (LSOR) method where the points on one row of the mesh are grouped together (Varga [1962]). It has been proved that for the solution of the Dirichlet problem with the uniform mesh size h in the unit square (see Section 4.6), the LSOR method is asymptotically faster by a factor of $\sqrt{2}$ over the point Successive Over-Relaxation (SOR) method (Parter [1981]).

In this chapter, a method involving a block of four points is implemented in parallel with the acceleration parameter, $\omega$, where the principle theory of the LSOR is valid for this method as well. However, many parallel implementations of the 4-point block method are developed in this chapter. These include synchronous and asynchronous implementations. The scheme that is used to update any single component is exactly as that was used in the Purely Asynchronous (PA) method of Chapter 4, Section 4.10. This means that the component is updated as soon as it is evaluated and the processors are never blocked or need wait for other processors.

This scheme is used in the algorithms of this chapter in order to obtain a basic framework to compare the 4-point block method with the standard point method which was represented in the PA method. The comparisons are based upon the number of operations performed at each mesh point as well as the number of iterations which is obtained experimentally.

The basic derivation of the sequential 4-point method is presented
as well as the performance analysis of the different implementations. The principle behind the analysis is that parallel computing involves the sharing of some resources which have a limited availability. This has the consequence that there is a limit to the number of demands that can be satisfied and some of them must wait if there are some competing ones. The demands are determined by the program while the availability and allocation algorithm are properties of the system. The performance of a given algorithm on a given parallel system can be obtained by analysing the properties of the system under various theoretical demand patterns. This analysis is substantiated by an analysis of the resources provided by a particular computer system, and the resources demanded by some parallel programs. The results of the different implementations of the 4-point block method are presented in this chapter and analysed theoretically to yield predicted results which are then compared with the actual results when these algorithms are run on the NEPTUNE system.
5.2 **THE BASIC FOUR-POINT BLOCK ITERATIVE METHOD**

The Four-Point Block iterative method was developed by Evans and Biggins [1982] and applied to solve the model problem of the solution of the Laplace equation in the unit square. In this method, the mesh points are ordered in groups of four points and the groups themselves are ordered in red-black ordering as shown in Figure 5.1.

![Figure 5.1](image)

**FIGURE 5.1**

Suppose that the system of equations to be solved by the two-dimensional Dirichlet problem (see Section 4.6), where the five-point scheme shown in Figure 5.2 is used is given by the form,

$$A x = b$$

(5.2.1)

The left-hand side of the finite difference equation of such a system has the form

$$x_{i,j} + a_1 x_{i-1,j} + a_2 x_{i,j+1} + a_3 x_{i+1,j} + a_4 x_{i,j-1}.$$

![Figure 5.2](image)
The coefficient matrix $A$ of equation (5.2.1) which is illustrated below for an 8 x 8 mesh has the block structure

$$A = \begin{bmatrix}
R_0 & R_0 & \cdots & R_0 & 0 & \cdots & 0 & R_0 \\
R_0 & R_0 & \cdots & R_0 & 0 & \cdots & 0 & R_0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & R_4 & R_2 & \cdots & R_4 & R_0 \\
0 & 0 & \cdots & R_4 & R_2 & \cdots & R_4 & R_0 \\
\end{bmatrix}$$

where

$$R_0 = \begin{bmatrix} 1 & \alpha_2 & 0 & \alpha_3 \\ \alpha_4 & 1 & \alpha_3 & 0 \\ 0 & \alpha_1 & 1 & \alpha_4 \\ \alpha_1 & 0 & \alpha_2 & 1 \\ \end{bmatrix}, \quad R_1 = \begin{bmatrix} 0 & 0 & 0 & \alpha_1 \\ 0 & 0 & \alpha_1 & 0 \\ 0 & 0 & 0 & 0 \\ \end{bmatrix}$$

$$R_2 = \begin{bmatrix} \alpha_2 & 0 & 0 & 0 \\ 0 & \alpha_2 & 0 & 0 \\ 0 & 0 & \alpha_2 & 0 \\ \end{bmatrix}, \quad R_3 = \begin{bmatrix} \alpha_3 & 0 & 0 & 0 \\ 0 & \alpha_3 & 0 & 0 \\ 0 & 0 & \alpha_3 & 0 \\ \end{bmatrix}$$

and

$$R_4 = \begin{bmatrix} 0 & \alpha_4 & 0 & 0 \\ 0 & 0 & \alpha_4 & 0 \\ 0 & 0 & 0 & \alpha_4 \\ \end{bmatrix}$$
As can be seen, the matrix $A$ here is block partitioned. However, since the blocks are taken in red-black ordering, then the matrix has block property (A) and is block consistently ordered. Also, if the blocks are taken in natural ordering, the matrix is block consistently ordered. Therefore, for both orderings, the full theory of block SOR is applicable to this method.

In the point iterative methods discussed in section (4.3.2) each component of the iterate $x^{(k)}$ is determined explicitly. This means, it can be calculated by itself using the values of the components which are evaluated previously. Whereas, in the block iterative method, one can improve the values of the approximate solution simultaneously with a group of points on the mesh. Such methods are called implicit methods since the solution of a whole group of points can be found at a time. In fact, implicit methods have larger convergence rates, at the cost of some extra computation in each iteration than that of the explicit method.

The explicit block SOR method can be considered as a point SOR method applied to a transformed matrix, $A^E$. However, it is necessary to calculate the matrix

$$
A^E = (\text{diag}(R_0))^{-1} A.
$$

The matrix $(\text{diag}(R_0))^{-1}$ is simply $\text{diag}(R_0^{-1})$ and $R_0^{-1}$ is given by

\[
R_0^{-1} = \begin{bmatrix}
1-a_1a_3-a_2a_4 & a_2(a_2a_4-a_1a_3-1) & 2a_2a_3 & a_3(a_1a_3-a_2a_4-1) \\
a_4(a_2a_4-a_1a_3-1) & 1-a_1a_3-a_2a_4 & a_3(a_1a_3-a_2a_4-1) & 2a_3a_4 \\
2a_1a_4 & a_1(a_1a_3-a_2a_4-1) & 1-a_1a_3-a_2a_4 & a_4(a_2a_4-a_1a_3-1) \\
a_1(a_1a_3-a_2a_4-1) & 2a_1a_2 & a_2(a_2a_4-a_1a_3-1) & 1-a_1a_3-a_2a_4
\end{bmatrix}
\]

\[
\text{where,} \quad d = (a_1a_3-a_2a_4)^2 -2(a_1a_3+a_2a_4) + 1. \tag{5.2.3}
\]

The block structure of $A^E$ is the same as that of the matrix $A$ given in equation (5.2.2a) with the submatrices $R_0$ replaced by identity matrices.
I, and the submatrices $R_i, i=1,2,3,4$ replaced by $R_0^{-1}R_i$. As $R_i$ has a column of zeros, so does $R_0^{-1}R_i$, and since an element $a_{ij}$ occurs as the $(p,q)^{th}$ element of $R_i$, the $q$th column of $R_0^{-1}R_i$ is the $p$th column of $R_0^{-1}$, multiplied by $a_{ij}$. For example, for our model problem, the Dirichlet problem

$$a_1 = a_2 = a_3 = a_4 = -\frac{1}{4},$$

so that from (5.2.3) we have,

$$R_0^{-1} = \frac{1}{6}
\begin{bmatrix}
7 & 2 & 1 & 2 \\
2 & 7 & 2 & 1 \\
1 & 2 & 7 & 2 \\
2 & 1 & 2 & 7
\end{bmatrix}
$$

and, for example,

$$R_0^{-1}R_1 = -\frac{1}{24}
\begin{bmatrix}
0 & 0 & 2 & 7 \\
0 & 0 & 7 & 2 \\
0 & 0 & 2 & 1 \\
0 & 0 & 1 & 2
\end{bmatrix}
$$

Now to derive the corresponding equations of the points that form a block of four points, consider the area of a mesh shown in Figure 5.3.

![Figure 5.3](image)

The main difference between the block method and the point method is that in the former method, the equations of all the points in the block are formed in a way that the points are not dependent on each other in obtaining the new values. However, the equations required for the calculation of the values of $x_1, x_2, x_3$ and $x_4$ of Figure 5.3 contain certain
expressions that involve $X_A, X_B, \ldots, X_H$ is common, and these are only calculated once and stored. Thus, if we let

$$s_1 = X_A + X_B,$$
$$s_2 = X_C + X_D,$$
$$s_3 = X_E + X_F,$$
$$s_4 = X_G + X_H,$$
$$s_5 = s_1 + s_3 + s_3,$$
$$s_6 = s_2 + s_2 + s_4 + s_4,$$

then we have,

$$x_1 = \frac{1}{24} (7s_1 + s_6 + s_3),$$
$$x_2 = \frac{1}{24} (7s_2 + s_5 + s_4),$$
$$x_3 = \frac{1}{24} (7s_3 + s_6 + s_1),$$
$$x_4 = \frac{1}{24} (7s_4 + s_5 + s_2).$$

When the over-relaxation factor, $\omega$, is added, these equations take the following form if $x_1$ for example, is considered:

$$x^{(k+1)}_1 = x^{(k)}_1 + \omega (x^*_1 - x^{(k)}_1),$$

where $x^*_1$ is the value of $x_1$ in equation (5.2.5) above.

Equation (5.2.6) requires an average of 6$\frac{1}{2}$ additions and 3 multiplications per point per iteration with the assumption that the constant $\frac{1}{24}$ is stored. This calculation can be less if the equations in (5.2.4) and (5.2.5) are re-organised. Thus, the explicit form of this block SOR method takes only about as much work as the implicit form of the 4 point method (Biggins [1980]).

Finally, we present the formula for the over-relaxation factor $\omega$ for the Four-point block method. It is mentioned above, that, the theory for this method is equivalent to the theory of the SLOR method (Parter, 1981).
Therefore,
\[ \omega_{\text{opt}} = \frac{2}{1 + \sqrt{1 - \rho^2(B)}} \]  
(5.2.7)

where \( \rho(B) \) is the spectral radius of the Jacobi iteration matrix \( B \) that corresponds to the four-point block iterative method. In actual fact, the spectral radius \( \rho(B) \) cannot be obtained theoretically. Rather, it can be estimated experimentally. Thus Biggins [1980] has obtained the value of this quantity experimentally using the power method where the results turned out to be equivalent to the values obtained for the line Jacobi method. The spectral radius of the line Jacobi matrix is given by Varga [1962] as,
\[ \rho(B) = \frac{\cos \theta}{2 - \cos \theta} \]  
(5.2.8)

Thus, \( \omega_{\text{opt}} \) of equation (5.2.7) can now be easily obtained. Then, the spectral radius and the rate of convergence of this method can be obtained which are equivalent to that of the SLOR method. Thus the spectral radius of the SLOR matrix \( (L^')_\omega \) is given by,
\[ \lambda' = \omega_{\text{opt}} - 1 \]  
(5.2.9)

and the rate of convergence of SLOR is
\[ R(L^')_\omega \sim 2 \sqrt{2 \rho(B)} \]  
(5.2.10)

For our model problem, the rate of convergence of the SLOR method is given by Varga [1962] as
\[ R(L^')_\omega \sim \frac{2 \sqrt{2 \pi}}{m+1} \text{, } m \to \infty \]  
(5.2.11)

This completes the basic principles of the four-point block iterative method from which we have derived some parallel versions and run them on the NEPTUNE system.
5.3 EXPERIMENTAL RESULTS OF THE PARALLEL FOUR-POINT BLOCK ITERATIVE METHOD

After presenting the basic concepts of the four-point block iterative method, we implement the method in parallel where many strategies are used to solve the model problem, the two-dimensional Dirichlet problem by using equations (5.2.4), (5.2.5) and (5.2.6). The strategies are mainly concerned with the way the problem to be solved is decomposed into many subsets that can be run in parallel. Clearly, different techniques may yield different results in the running-times and in the speed-up ratios. However, interesting results are obtained when the problem to be solved is programmed both synchronously and asynchronously.

We discuss in this section three strategies that we have developed where each strategy involves synchronous and asynchronous versions. The study includes comparisons between the synchronous and asynchronous versions in each strategy as well as comparisons between the results of different strategies. In all strategies, the update of a component follows exactly the same scheme of the Purely Asynchronous (PA) algorithm discussed in Chapter 4 (section 4.9.1) where no critical section is used. The algorithms of these strategies were run for $\omega=1.0$, i.e. similar to the Gauss-Seidel point method, and for $\omega=\omega_{opt}$, i.e. similar to the SOR point iterative method.

First Strategy

In the first strategy, the four-point block method is implemented so that a number of processes (paths) equal to the number of co-operating processors is generated where each path works on a subset of lines $N_s=N/P$ where $N$ is the number of rows in the mesh (divisible by $P$) and $P$ is the number of co-operating processors. This means that $P$ subsets are formed where each subset contains $N_s$ rows of the mesh. In this strategy, $N_s$
should also be a multiple of 2. Each processor then computes its own subset by taking up each successive two neighbouring rows at a time so that each block on these two lines is evaluated, i.e., in the natural ordering. When the blocks on these two lines are all evaluated, the next two lines are taken and the algorithm proceeds as before until all the lines in the subset are evaluated.

This strategy is implemented synchronously and the program involved is denoted by Prog 5.1. In this program, each processor evaluates its own subset in the manner discussed above and synchronises itself after each iteration step. When all the processors are synchronised, the convergence test is performed by one processor (the master processor usually) and if all the components of the mesh are obtained with the required accuracy then the procedure terminates otherwise a new cycle is repeated and so on until all the components have converged. The results of Prog 5.1 are listed in Table 5.1 where (12×12) and (16×16) mesh sizes were used in the experiments with \(\omega=1.0\) and \(\omega=\text{optimum } \omega\) which equals to 1.51 in the case of the (12×12) mesh and (1.60) in the case of the (16×16) mesh.

The same synchronous strategy is again implemented in Prog 5.2, but the blocks on each two lines are evaluated in the red-black ordering, i.e., blocks numbered as 1,3,5,... are taken first then the even numbered blocks 2,4,6,... . The results of this program are listed in Table 5.2 where the (16×16) mesh size is used only.

From the results of Tables 5.1 and 5.2, we notice that the running-time of Prog 5.2 is greater than that of Prog 5.1 and the speed-up for the natural ordering of Prog 5.1 is higher than the speed-up of the red-black ordering of Prog 5.2. Therefore, we choose the natural ordering scheme amongst the two implementations for the asynchronous version of this strategy.
<table>
<thead>
<tr>
<th>Mesh size (N×N)</th>
<th>P</th>
<th>ω</th>
<th>Time (seconds)</th>
<th>No. of iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12×12)</td>
<td>1</td>
<td>1.0</td>
<td>89.00</td>
<td>89</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>48.490</td>
<td>88</td>
<td>1.84</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.0</td>
<td>32.880</td>
<td>88</td>
<td>2.71</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.51</td>
<td>21.100</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.51</td>
<td>11.140</td>
<td>20</td>
<td>1.89</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.51</td>
<td>7.590</td>
<td>20</td>
<td>2.78</td>
</tr>
<tr>
<td>(16×16)</td>
<td>1</td>
<td>1.0</td>
<td>256.510</td>
<td>144</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>138.690</td>
<td>143</td>
<td>1.85</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>70.920</td>
<td>143</td>
<td>3.62</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.60</td>
<td>47.860</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.61</td>
<td>25.390</td>
<td>26</td>
<td>1.89</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.60</td>
<td>13.150</td>
<td>26</td>
<td>3.64</td>
</tr>
</tbody>
</table>

**TABLE 5.1:** The results of Prog 5.1, the synchronous 4-point block method, 2 lines are taken at a time (natural ordering)
The asynchronous version in this strategy was implemented in the same manner as in the synchronous version so that the blocks were taken in natural ordering. This scheme is programmed in Prog 5.3 where each subset of lines is allocated to a processor that runs asynchronously on its subset without waiting for other processors to complete their computations. In this case, each processor iterates permanently on its subset as in the Purely Asynchronous algorithm, (see Section 4.9.1), until this subset as well as the subsets that are carried out by other processors have converged. Therefore, a flag is assigned to each processor, where the set of all flags is in the shared memory and can be accessed by all processors, in order to check whether all the subsets have converged or not. The results of this implementation are listed in Table 5.3.

If we only consider the speed-ups obtained from Prog 5.1 and Prog 5.3, we notice that the asynchronous version is faster than the synchronous one for this algorithmic strategy. Moreover, from Tables 5.1 and 5.3, we see that the qualitative trend of the asynchronous results are slightly better than the corresponding synchronous results because of the synchronisation overheads, inspite of the efficient implementation on the NEPTUNE system. However, since each component is updated as soon as it is evaluated and made available to all the other processors which always obtain the most recent values of the components all the time, therefore the overheads obtained from synchronising the processes each cycle might not be very high. The list of Prog 5.3 is shown in Program A.5 in Appendix A.

Now, we compare the results of this asynchronous method for the explicit 4-point block method when with \( w=1.0 \) and \( w=w_{opt} \), and the Purely Asynchronous algorithm when run with \( w=1.0 \) and \( w=w_{opt} \). The timing results of the PA method are given in Table 5.4a. The factor of improvement of this method over the point iterative method implemented in the Purely
<table>
<thead>
<tr>
<th>Mesh size (N×N)</th>
<th>P</th>
<th>ω</th>
<th>Time (seconds)</th>
<th>No.of iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(16×16)</td>
<td>1</td>
<td>1.0</td>
<td>272.960</td>
<td>146</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>150.880</td>
<td>146</td>
<td>1.81</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>78.300</td>
<td>146</td>
<td>3.49</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.60</td>
<td>52.240</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.60</td>
<td>28.990</td>
<td>28</td>
<td>1.80</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.60</td>
<td>15.010</td>
<td>28</td>
<td>3.48</td>
</tr>
</tbody>
</table>

**TABLE 5.2:** The results of Prog 5.2, Synchronous 4-point block method (red-black ordering)

<table>
<thead>
<tr>
<th>Mesh size (N×N)</th>
<th>P</th>
<th>ω</th>
<th>Time (seconds)</th>
<th>No.of iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12×12)</td>
<td>1</td>
<td>1.0</td>
<td>88.470</td>
<td>89</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>46.340</td>
<td>90</td>
<td>1.91</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.0</td>
<td>32.310</td>
<td>93</td>
<td>2.74</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.51</td>
<td>20.950</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.51</td>
<td>11.00</td>
<td>21</td>
<td>1.90</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.52</td>
<td>7.450</td>
<td>22</td>
<td>2.81</td>
</tr>
<tr>
<td>(16×16)</td>
<td>1</td>
<td>1.0</td>
<td>254.920</td>
<td>144</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>132.750</td>
<td>145</td>
<td>1.92</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>68.530</td>
<td>145</td>
<td>3.72</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.60</td>
<td>47.820</td>
<td>27</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.61</td>
<td>24.990</td>
<td>27</td>
<td>1.91</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.61</td>
<td>13.050</td>
<td>28</td>
<td>3.66</td>
</tr>
</tbody>
</table>

**TABLE 5.3:** Prog 5.3 results, the asynchronous 4-point block method (2 lines at a time)
Asynchronous algorithm is given in Table 5.4b as $\gamma$, where,

$$
\gamma = \frac{\text{Time for the point iterative method (PA)}}{\text{Time for the 4-point block method}}
$$

From Table 5.4b, we see that the factor is large for the case $\omega=1.0$ which means that the 4-point block method is about 1.5 faster than the PA method. This is not true for $\omega=\omega_{\text{opt}}$, i.e. when we have the SOR version, because this method does not show a very large factor over the PA point method. In fact, the 4-point block method is faster than the PA point method by more than $\sqrt{2}$, whereas, the 4-point block SOR method is faster than the PAOR point method about $(\sqrt{2})^{1/2}$.

The run-time results for the (16x16) mesh size of the PA method of Table 5.4a and that of the asynchronous 4-point block method are illustrated in Figure 5.4. The speed-up ratios of both methods are also illustrated in Figure 5.5.

**Second Strategy**

In this strategy, the mesh lines are grouped into subsets where each subset contains $N_s$ lines as in the first strategy. In this strategy, the lines of a subset are taken one at a time, therefore, the first two components of each block are evaluated when the line holding them is taken by a processor. When all components on this line are evaluated, the next line that holds the other two components of each block is to be taken. The natural ordering scheme is used to evaluate the half blocks on each line.

This strategy has also been implemented in both synchronous and asynchronous versions. The synchronous version was implemented in Prog. 5.4 and the asynchronous version in Prog 5.5. The iteration cycles and the convergence test for both versions were implemented as in the first strategy. The results of Prog 5.4 and Prog 5.5 which are obtained from the NEPTUNE system, are listed in Tables 5.5 and 5.6 respectively.
<table>
<thead>
<tr>
<th>Mesh size</th>
<th>P</th>
<th>( \omega )</th>
<th>( \gamma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12x12)</td>
<td>1</td>
<td>1.0</td>
<td>1.55</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \omega_{opt} )</td>
<td>1.13</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>1.51</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \omega_{opt} )</td>
<td>1.18</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.0</td>
<td>1.46</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \omega_{opt} )</td>
<td>1.08</td>
</tr>
<tr>
<td>(16x16)</td>
<td>1</td>
<td>1.0</td>
<td>1.54</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \omega_{opt} )</td>
<td>1.16</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>1.56</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \omega_{opt} )</td>
<td>1.22</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>1.53</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \omega_{opt} )</td>
<td>1.09</td>
</tr>
</tbody>
</table>

**TABLE 5.4b:** The improvement factor obtained from the 4-point block method (Prog 5.3) over the PA point method

<table>
<thead>
<tr>
<th>Mesh size</th>
<th>P</th>
<th>( \omega )</th>
<th>Time</th>
<th>No. of iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12x12)</td>
<td>1</td>
<td>1.0</td>
<td>136.800</td>
<td>162</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>69.930</td>
<td>162</td>
<td>1.96</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.0</td>
<td>47.210</td>
<td>163</td>
<td>2.90</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>36.010</td>
<td>164</td>
<td>3.80</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.62</td>
<td>26.270</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.62</td>
<td>13.840</td>
<td>29</td>
<td>1.90</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.62</td>
<td>8.980</td>
<td>28</td>
<td>2.93</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.62</td>
<td>6.650</td>
<td>28</td>
<td>3.95</td>
</tr>
<tr>
<td>(16x16)</td>
<td>1</td>
<td>1.0</td>
<td>393.840</td>
<td>263</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>207.010</td>
<td>265</td>
<td>1.90</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>105.090</td>
<td>270</td>
<td>3.75</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.70</td>
<td>55.600</td>
<td>37</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.70</td>
<td>29.460</td>
<td>38</td>
<td>1.89</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.70</td>
<td>15.130</td>
<td>38</td>
<td>3.70</td>
</tr>
</tbody>
</table>

**TABLE 5.4a:** The results of the PA point method of Chapter 4
FIGURE 5.4
The Timing Results of The PA and The 4-Point Block Methods For (16X16) Mesh Size
FIGURE 5.5

The Speed-up Ratios For The PA and The Asynchronous 4-Point Block(Prog5.3) Methods
<table>
<thead>
<tr>
<th>Mesh Size</th>
<th>P</th>
<th>ω</th>
<th>Time (seconds)</th>
<th>No. of iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12×12)</td>
<td>1</td>
<td>1.0</td>
<td>104.730</td>
<td>89</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>57.660</td>
<td>89</td>
<td>1.82</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.0</td>
<td>38.930</td>
<td>89</td>
<td>2.69</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>29.640</td>
<td>89</td>
<td>3.53</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.52</td>
<td>27.210</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.53</td>
<td>13.740</td>
<td>21</td>
<td>1.98</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.53</td>
<td>9.310</td>
<td>21</td>
<td>2.92</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.53</td>
<td>7.130</td>
<td>21</td>
<td>3.82</td>
</tr>
<tr>
<td>(16×16)</td>
<td>1</td>
<td>1.0</td>
<td>304.720</td>
<td>145</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>166.110</td>
<td>145</td>
<td>1.83</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>83.430</td>
<td>144</td>
<td>3.64</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.61</td>
<td>62.920</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.61</td>
<td>33.340</td>
<td>29</td>
<td>1.89</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.61</td>
<td>16.340</td>
<td>28</td>
<td>3.85</td>
</tr>
</tbody>
</table>

**TABLE 5.5:** The results of Prog 5.4, the synchronous 4-point method (taken 1 line at a time)

<table>
<thead>
<tr>
<th>Mesh Size</th>
<th>P</th>
<th>ω</th>
<th>Time (seconds)</th>
<th>No. of iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12×12)</td>
<td>1</td>
<td>1.0</td>
<td>104.680</td>
<td>89</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>56.090</td>
<td>91</td>
<td>1.87</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.0</td>
<td>37.780</td>
<td>92</td>
<td>2.77</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>28.310</td>
<td>92</td>
<td>3.70</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.53</td>
<td>27.150</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.53</td>
<td>14.200</td>
<td>23</td>
<td>1.91</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.54</td>
<td>9.640</td>
<td>23</td>
<td>2.84</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.54</td>
<td>7.140</td>
<td>23</td>
<td>3.80</td>
</tr>
<tr>
<td>(16×16)</td>
<td>1</td>
<td>1.0</td>
<td>303.480</td>
<td>145</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>159.570</td>
<td>146</td>
<td>1.90</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>80.140</td>
<td>146</td>
<td>3.79</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.61</td>
<td>62.870</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.62</td>
<td>33.620</td>
<td>31</td>
<td>1.87</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.62</td>
<td>16.370</td>
<td>30</td>
<td>3.84</td>
</tr>
</tbody>
</table>

**TABLE 5.6:** The results of Prog 5.5, the asynchronous 4-point method (taken 1 line at a time)
From the results in Tables 5.5 and 5.6, we see that the synchronous and asynchronous versions of this strategy require approximately similar running-times when only one processor was in use, whereas for 2 or more processors the asynchronous version required slightly less time than the synchronous version. However, the speed-up ratios for the asynchronous version when $\omega=1.0$ are greater than that of the synchronous version, while the converse is true for $\omega=\omega_{opt}$. Also, the optimum $\omega$ is slightly different for both versions.

By comparing the results of the asynchronous versions of the first and second strategies as illustrated in Tables 5.3 and 5.6 respectively, we see that the strategy of uptaking two lines at a time requires less time than that of uptaking one line at a time. This is mainly related to the fact that in the strategy of uptaking two lines at a time the whole block is evaluated at once and more recent values of components are available for use. While, in the strategy of uptaking one line at a time, only half of the block at a time is evaluated and the second half is evaluated after the evaluation of the whole line is completed, and this naturally involves some extra time.

Generally speaking, in the two above-mentioned strategies, it does not really matter whether the algorithm is synchronous or not as far as the timing results are concerned. This is because these algorithms were implemented so that each component is updated as soon as it is evaluated and then it is made available to all other processors which means using only one shared array to hold the component's values. However, in the synchronous versions each processor allocates a subset of lines and the processors are synchronised at the end of each iteration cycle to insure that the subsets that are carried out by other processors iterate on the new values of the components. This, of course, gives the correct
approximation to the solution of the linear system of equations with a fixed number of iterations for any number of cooperating processors. Although, the asynchronous versions are implemented in the same manner except that each processor iterates on its subset and never waits for other subsets to be completed, no gain is obtained in the time and this is explained by the first reason mentioned above.

As the decomposition used in both strategies generates equal disjoint subsets to be carried out by different processors, therefore, the complexity of evaluating any component by any processor is the same. However, the computational complexity of a component update is only 9 operations when the mesh size is (16×16), therefore, (16×9) operations are completed along each single line. If the subsets consist of \(N_p\) lines then \((N_p \times 16 \times 9)\) operations are to be performed by each processor. This is true for the synchronous and asynchronous algorithms but the synchronisation at the end of each iteration cycle and consequently the generation of the parallel paths at every iteration cycle generates some overheads that reduce the timing results by approximately 1 second in these implementations.

Since the synchronous form makes a number of accesses to a parallel path equal to the number of iterations that are required to satisfy the convergency test, therefore, this overhead generates a significant overhead on a MIMD computer. It follows that the use of the asynchronous version of the above strategies could be better suited for the MIMD computer, because the asynchronous algorithms do not generate a significant amount of "Parallel path access" overheads.

**Third Strategy**

The third strategy is completely different from the other two strategies in the manner of distributing the blocks of 4 points onto the
processors. The asynchronous version of this strategy is implemented in Program A.6 in Appendix A. The implementation was completed as follows:

Following the work done by Barlow and Evans [1982]. The row and column indices of the blocks of 4-points are stored in a shared list of two dimensions. The arrangement of the blocks in the list is represented by the red-black ordering. A shared index to the column of the list is used. Since the number of the rows are stored in the first row of the list and the columns in the second row, therefore, updating the shared index by a critical section will allow only one processor to evaluate the block which corresponds to that index. For the convergence test, a list of the number of flags that is equal to the number of blocks in the linear system of equations is used so that the flag for the block under consideration is set to 1 if any single component of the block has not converged, otherwise the flag is set to 0. In order to iterate again, each processor tests all of the flags of all blocks and if it finds any of them sets to 1 then it iterates again choosing any available block. Otherwise, a global flag is set indicating that the convergence has been achieved for all components in the mesh. Since setting the global flag is performed by one processor which terminates its path afterwards, the other processors will only terminate when they attempt to uptake another block. The results of this implementation are listed in Table 5.7 where the program for it is called Prog 5.6.

In order to compare the asynchronous versions of the three strategies we list the timing results with the speed-up ratios of the three programs, Prog 5.3, Prog 5.5 and Prog 5.6 in Table 5.8, where a (16×16) mesh size is chosen. From Table 5.8, we notice that the asynchronous algorithm of the third strategy achieves better speed-up ratios than the other strategies in both cases when \( w=1.0 \) and \( w\leq w_{opt} \). These speed-ups
<table>
<thead>
<tr>
<th>Mesh Size (N×N)</th>
<th>p</th>
<th>ω</th>
<th>Time (seconds)</th>
<th>Number of Iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(16×16)</td>
<td>1</td>
<td>1.0</td>
<td>271.830</td>
<td>146</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>138.980</td>
<td>147</td>
<td>1.96</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>93.100</td>
<td>148</td>
<td>2.92</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>70.820</td>
<td>150</td>
<td>3.84</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.61</td>
<td>52.120</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>26.720</td>
<td>29</td>
<td>1.95</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>17.880</td>
<td>30</td>
<td>2.92</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>13.620</td>
<td>31</td>
<td>3.83</td>
</tr>
</tbody>
</table>

**TABLE 5.7**: The results of Prog 5.6, the asynchronous 4-point block method (the list of blocks)
<table>
<thead>
<tr>
<th>Method</th>
<th>( \omega )</th>
<th>( P )</th>
<th>Time</th>
<th>No. of Iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Prog 5.3</td>
<td>1.0</td>
<td>1</td>
<td>254.920</td>
<td>144</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>132.750</td>
<td>145</td>
<td>1.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>68.530</td>
<td>145</td>
<td>3.72</td>
</tr>
<tr>
<td>(2) Prog 5.5</td>
<td>1.0</td>
<td>1</td>
<td>303.480</td>
<td>145</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>159.570</td>
<td>146</td>
<td>1.90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>80.140</td>
<td>146</td>
<td>3.79</td>
</tr>
<tr>
<td>(3) Prog 5.6</td>
<td>1.0</td>
<td>1</td>
<td>271.830</td>
<td>146</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>138.980</td>
<td>147</td>
<td>1.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>93.100</td>
<td>148</td>
<td>2.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>70.820</td>
<td>150</td>
<td>3.84</td>
</tr>
<tr>
<td>(1) Prog 5.3</td>
<td>1.60</td>
<td>1</td>
<td>47.820</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.61</td>
<td>2</td>
<td>24.990</td>
<td>27</td>
<td>1.91</td>
</tr>
<tr>
<td></td>
<td>1.61</td>
<td>4</td>
<td>13.050</td>
<td>28</td>
<td>3.66</td>
</tr>
<tr>
<td>(2) Prog 5.5</td>
<td>1.61</td>
<td>1</td>
<td>62.870</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.62</td>
<td>2</td>
<td>33.620</td>
<td>31</td>
<td>1.87</td>
</tr>
<tr>
<td></td>
<td>1.62</td>
<td>4</td>
<td>16.370</td>
<td>30</td>
<td>3.84</td>
</tr>
<tr>
<td>(3) Prog 5.6</td>
<td>1.61</td>
<td>1</td>
<td>52.120</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>26.720</td>
<td>29</td>
<td>1.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>17.880</td>
<td>30</td>
<td>2.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>13.620</td>
<td>31</td>
<td>3.83</td>
</tr>
</tbody>
</table>

**TABLE 5.8:** The results of the asynchronous versions of the three strategies
are mostly linear, i.e. of order $P$, where $P$ is the number of processors that are in use. However, if we consider the timing results in Table 5.8, we see that the algorithm of the first strategy (i.e. the 2-lines strategy) requires less time than the other two strategies, meanwhile, the algorithm of the second strategy (the 1-line strategy) requires more time than the others.

If we consider the results of Prog 5.3 and Prog 5.6, we see that the time in Prog 5.3 is much less than that of Prog 5.6 when $P=1$, but the difference in the timing results of the two algorithms becomes smaller when $P$ is greater than 1.

The difference in the running-times for the three algorithms is due to the fact that in taking up 2 lines at a time some time is saved since the two lines are evaluated in one go, where each block on these two lines is carried out at a time. This, of course, is different to taking up one line at a time as described before. On the other hand, by evaluating the blocks in natural order where the blocks are stored in a list which is accessible by all processors, may require extra time to update the index to that list of blocks and to pick up a block of 4 points to evaluate. This last method seems to be better amongst the other methods for implementation on more than one processor as it is obvious from the speed-up ratios of Table 5.8. For illustration, the speed-up ratios for the three strategies are represented in Figures 5.6 and 5.7 when $w=1.0$ and $w=\omega_{opt}$ respectively.

As in the first and second strategies, the synchronous version is also implemented in the third strategy. The implementation was in such a way that a number of parallel paths equal to the number of blocks in the mesh is generated and carried out by any available number of processors. This means that each path picks up a block different from the other paths and
The Speed-up of The Three Strategies When $w=1.0$
The Speed-up of The Three Strategies When $w = w_{opt}$
this is insured by updating a shared index to the list by a critical
section. However, all the paths are synchronised at the end of each cycle
of the iterate. In case where not all the blocks have converged, the
paths will be generated again where each path iterates on the available
recent values. This procedure is continued until all the blocks have
converged. Also, in this algorithm the blocks are arranged in red-black
ordering in the list. The program for this implementation is called Prog
5.7 whose results are listed in Table 5.9.

If we compare the results of the asynchronous and synchronous versions
of this strategy which are in Tables 5.7 and 5.9, respectively we see that
the time required by the asynchronous version is less than that of the
synchronous one and also the asynchronous version indicates a very much
higher speed-up over the synchronous version. The timing results of the
two versions are listed in Figure 5.8, for $\omega=1.0$. This, however, is
different from the other two strategies in which the synchronous and
asynchronous versions have approximately equal timing results. The reason
why the two versions in the third strategy are different in their timing
results will be explained later in this chapter.

<table>
<thead>
<tr>
<th>Mesh Size (N x N)</th>
<th>P</th>
<th>$\omega$</th>
<th>Time (Seconds)</th>
<th>No. of iterations</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(16 x 16)</td>
<td>1</td>
<td>1.0</td>
<td>296.770</td>
<td>146</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>159.460</td>
<td>146</td>
<td>1.86</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>108.410</td>
<td>146</td>
<td>2.74</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>86.910</td>
<td>146</td>
<td>3.41</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.61</td>
<td>56.970</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.61</td>
<td>30.310</td>
<td>28</td>
<td>1.88</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.61</td>
<td>20.680</td>
<td>28</td>
<td>2.75</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.61</td>
<td>16.570</td>
<td>28</td>
<td>3.44</td>
</tr>
</tbody>
</table>

**TABLE 5.9:** The results of Prog 5.7, the synchronous 4-point block
method (a list of blocks)
FIGURE 5.8
The Synchronous and Asynchronous Versions of The Third Strategy When w=1.0
The synchronous version (Prog 5.7) has some limitations. Since the number of parallel paths generated was equal to the number of blocks of 4 points, therefore, a linear system of equations of a large size which has more than 75 blocks of 4 points cannot be used since - at the time of the experiment - the NEPTUNE system can only generate 75 paths as a maximum limit including the path generating the parallel paths. Thus, for our model problem we run the problem for a (16x16) mesh size that contains 64 blocks. From another point of view, this synchronous algorithm generates a large amount of overheads, mainly the parallel path overheads. As each component of the mesh requires 9 arithmetic operations, therefore, (4x9) operations are required for a block of 4 points which is actually the computation required for each single path. This amount of computation is considered small when it is compared with the cost that is paid for synchronising all the paths. However, the synchronisation cost will be larger if the number of paths is greater than the number of available processors. This is because, the paths that complete their job remain idle for a long time whilst waiting for the other paths to complete their job. The same situation occurs as many times as the algorithm iterates to achieve convergence. Clearly, this generates a significant amount of overheads that degrade the performance of the algorithm.

On the other hand, in the asynchronous version (i.e. Prog 5.6), the number of parallel paths generated is equal to the number of cooperating processors where each path is iterated to evaluate each block without any synchronisation. Therefore, the asynchronous version is better than the synchronous one in this strategy, because it generates less overheads.

Finally, we compare the asynchronous version of the third strategy of the 4-point block method with the explicit point method that is implemented in the Purely Asynchronous method, the results of which are
listed in Table 5.4a above. Table 5.9a illustrates this comparison where
\( \gamma \) represents the ratio

<table>
<thead>
<tr>
<th>Mesh Size</th>
<th>P</th>
<th>( \omega )</th>
<th>( \gamma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(16x16)</td>
<td>1</td>
<td>1.0</td>
<td>1.45</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>&quot;</td>
<td>1.49</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>&quot;</td>
<td>1.48</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>( \omega_{\text{opt}} )</td>
<td>1.10</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>&quot;</td>
<td>1.10</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>&quot;</td>
<td>1.11</td>
</tr>
</tbody>
</table>

**TABLE 5.9a:** The improvement factor obtained from the asynchronous
form (a list of blocks) over the point method

From Table 5.9a, we notice that when \( \omega=1.0 \) the ratio \( \gamma \) becomes
approximately 1.5, whereas the ratio is slightly greater than unity when
\( \omega=\omega_{\text{opt}} \). This matches with the \( \gamma \) ratios of Table 5.4b that are shown to
be \( \sqrt{2} \) for \( \omega=1.0 \) and \( (\sqrt{2})^{1/4} \) for \( \omega=\omega_{\text{opt}} \).

In Figure 5.9, the timing results of the point version of the PA
method and the asynchronous versions of the three strategies of programs
Prog 5.3, Prog 5.5 and Prog 5.6, of the 4-point method are illustrated,
where a (16x16) mesh size is used for \( \omega=1.0 \) and \( \omega=\omega_{\text{opt}} \). We see the
difference in the timing results between the point method and the three
implementations of the 4-point block method, especially the first and
third strategies which possess less time than the PA method. This time
discrepancy is due to the computational work that is carried out by each
processor which, in fact, is different from one implementation to another.
This amount of work is calculated and compared with the equivalent part of
the synchronous version which possesses significant overheads and is shown
in the next section.
FIGURE 5.9
The Timing Results of The PA Method and The Asynchronous Forms of
The 3 Strategies of The 4-Point Block Method
5.4 PERFORMANCE ANALYSIS OF THE PARALLEL 4-POINT BLOCK METHODS AND THE STANDARD POINT METHOD

It was indicated, in the last section, how different algorithmic designs produce different timing results and speed-up ratios and also whether the implementation was done synchronously or asynchronously. In both cases the overhead measurements should be borne in mind and incurred when more than one processor is cooperating.

In fact, parallel computing requires three resources: multiple processors, communication for shared data, and synchronisation to ensure any necessary time ordering. Thus, parallel programs always require more than one processor, and there has to be some communication between the processors even if it is only as much as that required to start processing in the first instant.

The main feature in the analysis of the demand and supply of resources is that several demands may compete for the supply of a shared resource: i.e., processors, shared data structure or a memory block. This competition or contention has three consequences:

1. Since the shared resource has a limited availability it can satisfy only a finite number of demands in a finite time and this can limit the maximum performance of the program.

2. A mechanism is required to arbitrate between requests and keep all but one waiting, and this mechanism itself then imposes an overhead on the resource access even if there is no competition.

3. When requests contend, then all but one request will have to wait. The second and the third factors degrade the performance and these factors will be called respectively the static and dynamic costs of a shared resource access.

Barlow et al [1982b] discussed the performance analysis of the algorithms
on an asynchronous type machine and classified the sources of overheads as two types:

1. The static overhead which corresponds to the design of software and hardware. This includes the subdivision of the tasks, the allocation of these tasks to the available processors, and the checking by hardware and software for contention on accesses to the data base.

2. The dynamic overhead which corresponds to the interference between two or more subtasks running on different processors causing one or more of the processors to wait.

The performance of a multiprocessor can be expressed either in the form of a speed-up factor $S_p = \frac{T(1)}{T(P)}$, or in terms of the time wasted. However, the wasted time must be equal to the sum of the static and dynamic overheads. Let $W = P \times T(P) - T(1)$, be the wasted time which is the sum of times taken by the P processors to complete their subtasks minus the time taken on a uniprocessor. It is clear from this that either all processors complete processing together or some processors take longer than others. Thus,

$$T(P) \geq \frac{T(1) + W}{P}$$

It follows that,

$$S_p \leq \frac{PT(1)}{T(1) + W}.$$ 

Therefore, the maximum possible speed-up factor for a given algorithm can be determined by assuming that the dynamic overheads are zero. The dynamic overheads are zero only if every request for a resource occurs when that resource is not being used. This is true only if the demands for a resource are less than the supply of that resource.

In parallel computing, three resources are required: the processors, shared data, and the critical sections. We discuss now these three factors and their effect on a parallel system.
For the processors factor, the software measures the number of subtasks allocated to the processors and it counts the cycles that a processor is idle because there are no ready subtasks which are available to run.

For the shared data factor - this can be measured by counting the number of accesses to a shared data by going through the user's program.

Whilst for the critical regions - the software can measure the number of accesses made by a processor to a critical region and the number of cycles a processor remains idle because this resource is accessed by other processors.

Performance of the Algorithms

For the performance analysis of all the algorithms discussed in this chapter, we present the actual measurements obtained from the experiments on the NEPTUNE system as well as the measurements that are obtained manually from the program. These measurements indicate the total computational complexity performed by each path and the calculation of how many parallel paths and critical sections a single processor has been made. For our measurements we need to know the resource times in the NEPTUNE system. These are obtained from (Barlow, et al [1981]) and illustrated in Table 5.10. However, Table 5.11 illustrates the resource demands of the standard point method of the Purely Asynchronous method and the 4-point block method implemented synchronously or asynchronously for all three strategies discussed in the previous section. The term "flops" in these tables represents a floating point operation.

In fact, Table 5.11 gives the mean rate of accesses to the shared data, parallel paths scheduling and critical sections. It gives estimates of the potential speed-up from using P processors, where N represents the
number of rows in the mesh to be solved.

On the other hand, Table 5.12 illustrates the results obtained when the algorithms were run on the NEPTUNE system. The parallel paths and critical sections measurements are taken for the case of 4 processors. The parallel control access overheads and the shared data access overheads are taken for the case P=1. N was 16 in all experiments.

To obtain the parallel control access overheads and the shared data access overhead when the NEPTUNE system is used the same algorithm should be compiled using the following commands:

(i) the XPFCLS: this command generates a load module with no shared data assigned into the shared memory and no parallel path allocation. Thus, all the parallel programming constructs will be treated as ordinary Fortran constructs.

(ii) The XPFCLN: this command only loads the shared data into the shared memory. Therefore, by comparing the result of this command with results obtained from (i), we obtain the shared data access overheads.

(iii) The XPFCL: this command generates the load module as in (ii) in addition to the parallel path allocations. The comparison of these results with that of (ii) yields the parallel control overheads.

Now, by examining the Purely Asynchronous and the PAOR algorithms which are based on the standard point method, we see that a linear speed-up has been achieved and up to the number of the mesh points (i.e. \(N^2\)) processors can be employed as an upper limit. The PAOR algorithm has made seven accesses to the shared data per 11 floating point operations. From the results of Table 5.10, the shared data access timing is \(-0.7\) micro-second, \((0.7\) is the average of \((0.8, 0.5, 0.7, 0.7)\) of the 4 processors in Table 5.10), and \(-700\)\(\mu\)s for flops. Therefore, the static shared data access overheads in this algorithm is obtained as follows:
\[ \frac{1}{11} \times \frac{0.7}{700} \times 100 = 0.06\% \]

<table>
<thead>
<tr>
<th>Resource</th>
<th>Processor</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative speeds</td>
<td></td>
<td>1</td>
<td>1.04</td>
<td>1.01</td>
<td>0.98</td>
</tr>
<tr>
<td>Floating point *</td>
<td></td>
<td>~700</td>
<td>~700</td>
<td>~700</td>
<td>~700</td>
</tr>
<tr>
<td>Integer *</td>
<td></td>
<td>~20</td>
<td>~20</td>
<td>~20</td>
<td>~20</td>
</tr>
<tr>
<td>Local memory access *</td>
<td></td>
<td>0.6</td>
<td>0.6</td>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>Shared memory access *</td>
<td></td>
<td>(0.6+0.8)</td>
<td>(0.6+0.5)</td>
<td>(0.6+0.7)</td>
<td>(0.6+0.7)</td>
</tr>
<tr>
<td>Mutual exclusion mechanics *</td>
<td></td>
<td>~400</td>
<td>~400</td>
<td>~400</td>
<td>~400</td>
</tr>
<tr>
<td>Mutual exclusion blocked</td>
<td></td>
<td>~200</td>
<td>~200</td>
<td>~200</td>
<td>~200</td>
</tr>
<tr>
<td>Parallel path mechanism *</td>
<td></td>
<td>~800</td>
<td>~800</td>
<td>~800</td>
<td>~800</td>
</tr>
<tr>
<td>Parallel path blocked *</td>
<td></td>
<td>~400</td>
<td>~400</td>
<td>~400</td>
<td>~400</td>
</tr>
</tbody>
</table>

* times in microseconds.

**TABLE 5.10:** The resources time on the NEPTUNE system

Now, we concentrate our attention on the synchronous versions of the three strategies, and consider initially Prog 5.1 of the first strategy discussed in the previous section. This program made 1 access per \((12.5 * N * S)\) flops, where \(S\) is the number of lines in each process. This results in a parallel path access loss equal to \(~0.2\%\) since the parallel path mechanism requires \(~800\mu s\) (see Table 5.10). This loss is the same as the static parallel path access loss of Table 5.12.

In the case of the second strategy, Prog 5.4 made 1 access to a parallel path per \((15 * N * S)\) flops which generated \(~0.1\%\) static loss which is equal to the results obtained from the NEPTUNE system. Similarly, for Prog 5.7 of the third strategy which required 12 accesses per 27 flops
and this generates a static shared data overhead of 0.04%. The other losses in this program are the parallel control overheads which are the parallel path losses and the critical section losses. The program made 1 access per 53 flops to a parallel path and the same with the critical section mechanism. Therefore, from the results of Table 5.10, we obtain the parallel path static loss as 2% and a critical section loss as 1% which agree with the results of Table 5.12.

From the above-mentioned figures we conclude that the predicted results obtained from the synchronous program accesses to a critical region or a parallel path were in agreement to the experimental results which were obtained from the NEPTUNE system. These then are the static losses. The dynamic losses were obtained directly from the experimental results, and these are the contention for a parallel path or for a critical region.

The losses obtained from the synchronous algorithms in general represent some idle time in the total running-time which in many cases degrades the performance of the algorithms. This quantity is represented in Table 5.12.

The Idle Time

A processor can be idle in two cases. Firstly, if P parallel paths are not available at any time therefore the P processors cannot be processing and thus some of them must be idle. Secondly, all processors can be processing at any stage if the parallel paths can be allocated to processors taking into account the path's length and processor's speed for the particular set of data and the specific hardware involved. In this case, for different processors of different speeds, the faster processor can finish processing before the slower one and it will be forced to wait
<table>
<thead>
<tr>
<th>Program</th>
<th>Processors (P)</th>
<th>Shared Data</th>
<th>Parallel Path</th>
<th>Mutual Exclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Numbers</td>
<td>Speed-up</td>
<td>Access rate</td>
<td>overhead amount</td>
</tr>
<tr>
<td>1. PA (point method)</td>
<td>$PS^2N^2$</td>
<td>O(P)</td>
<td>7:11 flops</td>
<td>0.06%</td>
</tr>
<tr>
<td>2. Prog 5.7 (syn. list)</td>
<td>$PN^{-2}$</td>
<td>O(P)</td>
<td>12:27 flops</td>
<td>0.04%</td>
</tr>
<tr>
<td>3. Prog 5.6 (asyn. list)</td>
<td>$PN^{-2}$</td>
<td>O(P)</td>
<td>12:27 flops</td>
<td>0.04%</td>
</tr>
<tr>
<td>4. Prog 5.1 (syn. 2-lines)</td>
<td>$PN^{-2}$</td>
<td>O(P)</td>
<td>10:27 flops</td>
<td>0.037%</td>
</tr>
<tr>
<td>5. Prog 5.3 (asyn. 2-lines)</td>
<td>$PN^{-2}$</td>
<td>O(P)</td>
<td>10:27 flops</td>
<td>0.037%</td>
</tr>
<tr>
<td>6. Prog 5.4 (syn. 1-line)</td>
<td>$PSN$</td>
<td>O(P)</td>
<td>10:27 flops</td>
<td>0.037%</td>
</tr>
<tr>
<td>7. Prog 5.5 (asyn. 1-line)</td>
<td>$PSN$</td>
<td>O(P)</td>
<td>10:27 flops</td>
<td>0.037%</td>
</tr>
</tbody>
</table>

**TABLE 5.11**: Resource demands of algorithms [the standard point method and the 4-point block methods]
<table>
<thead>
<tr>
<th>Program</th>
<th>Idle time</th>
<th>Speed-up 2</th>
<th>Speed-up 3</th>
<th>Speed-up 4</th>
<th>( \omega )</th>
<th>Parallel path static contention</th>
<th>Critical regions static contention</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. PASYORl</td>
<td>-</td>
<td>1.9</td>
<td>-</td>
<td>3.8</td>
<td>1.0</td>
<td>0.002%</td>
<td>0.01%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.9</td>
<td>-</td>
<td>3.7</td>
<td>1.70</td>
<td>0.01%</td>
<td>0.03%</td>
</tr>
<tr>
<td>2. Prog 5.7</td>
<td>-5%</td>
<td>1.9</td>
<td>2.74</td>
<td>3.4</td>
<td>1.0</td>
<td>2%</td>
<td>1%</td>
</tr>
<tr>
<td>(syn.list)</td>
<td></td>
<td>1.9</td>
<td>2.75</td>
<td>3.4</td>
<td>1.61</td>
<td>2%</td>
<td>1%</td>
</tr>
<tr>
<td>3. Prog 5.6</td>
<td>-</td>
<td>1.96</td>
<td>2.92</td>
<td>3.84</td>
<td>1.0</td>
<td>0.002%</td>
<td>0.002%</td>
</tr>
<tr>
<td>(asyn.list of blocks)</td>
<td></td>
<td>1.96</td>
<td>2.92</td>
<td>3.83</td>
<td>1.61</td>
<td>0.002%</td>
<td>0.002%</td>
</tr>
<tr>
<td>4. Prog 5.1</td>
<td>-7%</td>
<td>1.85</td>
<td>-</td>
<td>3.62</td>
<td>1.0</td>
<td>0.2%</td>
<td>0.2%</td>
</tr>
<tr>
<td>(syn.2lines)</td>
<td></td>
<td>1.9</td>
<td>-</td>
<td>3.8</td>
<td>1.60</td>
<td>0.2%</td>
<td>0.2%</td>
</tr>
<tr>
<td>5. Prog 5.3</td>
<td>-</td>
<td>1.92</td>
<td>-</td>
<td>3.72</td>
<td>1.0</td>
<td>0.002%</td>
<td>0.03%</td>
</tr>
<tr>
<td>(asyn.2lines)</td>
<td></td>
<td>1.91</td>
<td>-</td>
<td>3.54</td>
<td>1.61</td>
<td>0.002%</td>
<td>0.07%</td>
</tr>
<tr>
<td>6. Prog 5.4</td>
<td>-7%</td>
<td>1.83</td>
<td>-</td>
<td>3.64</td>
<td>1.0</td>
<td>0.1%</td>
<td>0.2%</td>
</tr>
<tr>
<td>(syn.1line)</td>
<td></td>
<td>1.9</td>
<td>-</td>
<td>3.85</td>
<td>1.61</td>
<td>0.1%</td>
<td>0.2%</td>
</tr>
<tr>
<td>7. Prog 5.5</td>
<td>-</td>
<td>1.9</td>
<td>-</td>
<td>3.8</td>
<td>1.0</td>
<td>0.002%</td>
<td>0.02%</td>
</tr>
<tr>
<td>(asyn.1line)</td>
<td></td>
<td>1.9</td>
<td>-</td>
<td>3.84</td>
<td>1.62</td>
<td>0.01%</td>
<td>0.2%</td>
</tr>
</tbody>
</table>

**TABLE 5.12:** Performance measurements of algorithms on the NEPTUNE system
for the slower processor. This is obvious in the synchronous algorithms run on a MIMD computer.

It follows that if the speed of the processors is known, the amount of the idle time can be predicted by examining the complexity and the number of parallel paths run by each processor. However, from our previous algorithms the idle time loss in a run using 4 processors was measured and presented in Table 5.12 for the three types of synchronous algorithms.

Table 5.13 presents the complexity and the number of paths processed by each processor for the three programs. Hence, Prog 5.7 generated ~5% processor's idle time loss. This is obtained as follows:

Let $X = \text{total complexity per processor}$, thus

$$X = (\text{total number of operations}) \times (\text{time for 1 operation}) \times \frac{\text{Number of paths}}{}.$$ 

Then each processor's speed is multiplied by $X$ to obtain $P$ values, then the difference of (max. of the $P$ values, min. of the $P$ values) is obtained, let us call it $d$. Afterwards, we can obtain $I$, where

$$I = \frac{(d \times \text{number of iterations} \times 100)}{(\text{time of the algorithm})}.$$ 

The quantity $I$ represents the percentage ratio of the idle time corresponding to the total run time of the algorithm.

The other two synchronous programs have ~7% idle time loss. This is, however, slightly larger than that of the first program. The reason is as follows, if we check Table 5.13, we notice that the complexity per one path is (50 flops) for Prog 5.7, whereas it is (50 * 16 flops) for Prog 5.1 and (30 * 32 flops) for Prog 5.4. This means that the computational work per path is small in the algorithm of Prog 5.7 compared with the other two algorithms. Therefore, the processor performs a small amount of work and
TABLE 5.13: The complexity per path for the synchronous algorithms

<table>
<thead>
<tr>
<th>Program</th>
<th>$\omega$</th>
<th>complexity/ path</th>
<th>No. of paths per processor</th>
<th>Time</th>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Prog 5.7</td>
<td>1.0</td>
<td>1CS + S0 flops</td>
<td>16</td>
<td>86.910</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>1.61</td>
<td></td>
<td>16</td>
<td>16.570</td>
<td>28</td>
</tr>
<tr>
<td>2. Prog 5.1</td>
<td>1.0</td>
<td>50 flops<em>16</em>S</td>
<td>1</td>
<td>70.920</td>
<td>143</td>
</tr>
<tr>
<td></td>
<td>1.60</td>
<td></td>
<td>1</td>
<td>13.150</td>
<td>26</td>
</tr>
<tr>
<td>3. Prog 5.4</td>
<td>1.0</td>
<td>30 flops<em>16</em>S</td>
<td>1</td>
<td>83.430</td>
<td>144</td>
</tr>
<tr>
<td></td>
<td>1.61</td>
<td></td>
<td>1</td>
<td>16.340</td>
<td>28</td>
</tr>
</tbody>
</table>

*critical section.*

stops waiting for the slowest processor. This yields a short idle time, but since we run the algorithm for a (16x16) mesh size, then 64 parallel paths were generated and it is found that (16=64/4) paths were run by each processor when the run is made in 4 processors. On the other hand, for the other two algorithms, the computational work per path was large and when the processors were synchronised at the end of each iteration, the faster processor may wait for a long time for the slower processor to complete its work. This is the main reason for obtaining 7% idle time in these two algorithms whereas it was 5% in the Prog 5.7 although its complexity includes one access to a critical section which was a very small section.

The Difference Between the Standard Point Method and the 4-Point Explicit Block Method

Now, we compare the standard point method of the PA method and all the asynchronous versions of the three strategies to solve the 4-point block method in terms of their complexity per point which is performed by the algorithm.
Table 5.14 represents the complexity per point of the algorithms, the PA, Prog 5.3, Prog 5.5, and Prog 5.6. Since the time for the floating point operation is ~700μs, therefore the number of operations times the number of iterations is multiplied by 700. This will yield the total time for the complexity of the algorithm. These results together with the ratio $\gamma$, which is the ratio between the PA algorithm and any asynchronous algorithm of the three strategies of the 4-point block method, are illustrated in Table 5.15.

Although Table 5.14 indicates that the standard point method requires less operations per mesh point than that of the 4-point block method, the total time per mesh point is greater in the standard point method than that of the 4-point method as it is obvious from Table 5.15. This difference is because the standard point method (PA) requires more iterations than the other methods. The time discrepancy between the two different implementations is shown to be ~1.5 when $w=1.0$ and is slightly faster by a factor of ~1.1. From Table 5.15 we also notice that Prog 5.5, i.e. the strategy of taking up one line at a time is slower than the (PA) standard point method when $w=w_{opt}$. This is also clear from the diagrams illustrated in Figure 5.9.
<table>
<thead>
<tr>
<th>Program</th>
<th>Complexity per point</th>
<th>P</th>
<th>( \omega )</th>
<th>Experimental</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Time (CPU time)</td>
<td>Iterations</td>
</tr>
<tr>
<td>1. PA</td>
<td>10 flops</td>
<td>1</td>
<td>1.0</td>
<td>393.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.70</td>
<td>55.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1.0</td>
<td>207.01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.70</td>
<td>29.46</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.0</td>
<td>105.09</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.70</td>
<td>15.130</td>
</tr>
<tr>
<td>2. Prog 5.6</td>
<td>12.5 flops</td>
<td>1</td>
<td>1.0</td>
<td>271.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.61</td>
<td>52.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1.0</td>
<td>138.980</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.61</td>
<td>26.72</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.0</td>
<td>93.100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.61</td>
<td>17.98</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.0</td>
<td>70.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.61</td>
<td>13.620</td>
</tr>
<tr>
<td>3. Prog 5.3</td>
<td>12.5 flops</td>
<td>1</td>
<td>1.0</td>
<td>254.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.60</td>
<td>47.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1.0</td>
<td>132.75</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.61</td>
<td>24.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.0</td>
<td>68.53</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.61</td>
<td>13.570</td>
</tr>
<tr>
<td>4. Prog 5.5</td>
<td>15 flops</td>
<td>1</td>
<td>1.0</td>
<td>303.48</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.62</td>
<td>62.920</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1.0</td>
<td>159.570</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.62</td>
<td>33.62</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.0</td>
<td>80.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.62</td>
<td>16.370</td>
</tr>
</tbody>
</table>

**TABLE 5.14**: The complexity per point for the asynchronous algorithms
### TABLE 5.15: The comparison between the point method and the 4-point explicit block methods

<table>
<thead>
<tr>
<th>Program</th>
<th>P</th>
<th>$\omega$</th>
<th>Time for complexity/point</th>
<th>Total time</th>
<th>$\gamma$ ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA</td>
<td>1</td>
<td>1.0</td>
<td>0.007 sec</td>
<td>1.841</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.70</td>
<td>&quot;</td>
<td>0.259</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>&quot;</td>
<td>1.855</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.70</td>
<td>&quot;</td>
<td>0.266</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>&quot;</td>
<td>1.89</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.70</td>
<td>&quot;</td>
<td>0.266</td>
<td></td>
</tr>
<tr>
<td>Prog 5.6</td>
<td>1</td>
<td>1.0</td>
<td>0.00875</td>
<td>1.2775</td>
<td>1.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.61</td>
<td>&quot;</td>
<td>0.245</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>&quot;</td>
<td>1.28625</td>
<td>1.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.61</td>
<td>&quot;</td>
<td>0.25375</td>
<td>1.02</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.0</td>
<td>&quot;</td>
<td>1.295</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.61</td>
<td>&quot;</td>
<td>0.2625</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>&quot;</td>
<td>1.3125</td>
<td>1.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.61</td>
<td>&quot;</td>
<td>0.2725</td>
<td>0.99</td>
</tr>
<tr>
<td>Prog 5.3</td>
<td>1</td>
<td>1.0</td>
<td>0.00875</td>
<td>1.26</td>
<td>1.46</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.60</td>
<td>&quot;</td>
<td>0.23625</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>&quot;</td>
<td>1.26075</td>
<td>1.46</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.61</td>
<td>&quot;</td>
<td>0.2362</td>
<td>1.13</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>&quot;</td>
<td>1.295</td>
<td>1.46</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.61</td>
<td>&quot;</td>
<td>0.25375</td>
<td>1.1</td>
</tr>
<tr>
<td>Prog 5.5</td>
<td>1</td>
<td>1.0</td>
<td>0.01050</td>
<td>1.5225</td>
<td>1.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.62</td>
<td>&quot;</td>
<td>0.315</td>
<td>0.82</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.0</td>
<td>&quot;</td>
<td>1.533</td>
<td>1.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.62</td>
<td>&quot;</td>
<td>0.3255</td>
<td>0.82</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.0</td>
<td>&quot;</td>
<td>1.533</td>
<td>1.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.62</td>
<td>&quot;</td>
<td>0.3045</td>
<td>0.9</td>
</tr>
</tbody>
</table>
5.5 CONCLUSIONS

The three different strategies of the 4-point block iterative method present different timing results and different losses when they are run on the NEPTUNE system. For example, when the blocks of 4 points are arranged in a list as in the third strategy and the blocks were evaluated synchronously, the parallel control access overhead was 3%. When we calculate the number of accesses to a path per the number of operations per that path, the static loss was 0.2% which is the same as obtained from the experimental results (see Table 5.11 and 5.12). Actually, this strategy possesses the highest loss in the parallel control overhead if it is compared with the other two strategies.

In general, all the algorithms presented in this chapter have predicted static parallel path access overheads which agree with the overheads obtained when these algorithms were run on 4 processors on the NEPTUNE system. This is also true for the predicted overheads obtained from the mutual exclusion access rate (i.e. critical sections access rate) which only exist in the synchronous and asynchronous versions of the third strategy. The two Tables 5.11 and 5.12 show these agreements.

The measurements that were not predictable were the idle times arising from some processors having no paths to execute assuming that the processors have different speeds. The idle time obtained in the third strategy was less than what was obtained from the other two strategies when the synchronous algorithms are considered.

The first and third strategies used to implement the 4-point block explicit method, the 4-point block method had a factor of improvement of -1.5 over the standard point method when \( \omega = 1.0 \), and when \( \omega = \omega_{\text{opt}} \) this factor was approximately \( (\sqrt{2})^{\frac{1}{2}} \), although the strategies were different in their parallel implementations. However, the strategy of taking up one line at
a time was slower than the PA method when $\omega = \omega_{opt}$ for reasons discussed before. This is clear from the results of Table 5.15 and Figure 5.9.

Finally, as the amount of the idle time of the synchronous versions of the three strategies was varied from one strategy to another, they varied in their parallel implementations, we concluded that the implementation of the 4-point block method as in the third strategy, i.e. when the blocks of 4 points are arranged in red-black ordering in a list, was the best method amongst the other two implementations, and its asynchronous version had achieved a better speed-up which was almost linear with the number of processors available. This was because the processors were fully occupied and busy doing useful work most of the time.
CHAPTER 6

COMPARATIVE STUDIES OF PARALLEL SORTING AND MERGING ALGORITHMS
6.1 INTRODUCTION

Sorting is one of the most useful applications on computers in the sense that they make it possible to use sequential accessing on large files instead of using direct addressing. It is also considered as an aid to searching, therefore, it helps to make the computer output more suitable for human consumption. Traditionally, sorting has been used mostly for business data processing.

In the early 1950's there was some debate on terminology and a suggestion that there should be a distinction between (a) sorting in the sense of "distributing", and (b) sorting in the sense of "rearranging the order". This may be illustrated by the examples of (a) sorting letters into pigeon-holes and (b) re-arranging a list of authors to put them in alphabetical order. The distinction appears obvious in practical terms, but it is not of fundamental significance. In fact, many sorting algorithms have been developed since then and it was found that the time required to sort N records, using a competitive general-purpose sorting algorithm, is roughly proportional to (NlogN), where approximately log N "passes" over the data have to be made as in the quicksort algorithm (Knuth [1973]).

In recent years, many efficient implementations of comparison problems, such as sorting and merging using multiprocessor computing systems have attracted considerable attention. One of the earliest such implementations was due to Batcher [1968] who proposed a sorting network consisting of comparators based on the principle of iterated merging. Such a scheme sorts N keys with O(N(log N)^2) comparators in time O((log N)^2). Following Batcher's work, Muller and Preparata [1975] have shown that sorting N keys requires a time of O(log N), but it requires O(N^2) processors which is impractical.

Hirschberg [1978] developed algorithms suitable for SIMD-type parallel
computers with random access capabilities to a common memory and showed that N keys can be sorted in time $O(k \log N)$ with $N^{(1+1/k)}$ processors, where $k$ is any integer $\geq 2$. These algorithms are organised in such a way that the memory fetch conflicts, i.e. simultaneous reading of the same location by more than one processor, is possible. The problem of memory fetch conflicts had been considered by Preparata [1978] when he introduced an algorithm that sorts N keys with $N^{1+\alpha}$ processors, where $0<\alpha<1$, in time $(c/\alpha) \log N + O(\log N)$, for some constant $c$. Subsequently, new results were obtained on parallel merging by Gavril [1975]. Valiant [1975] also developed fast algorithms that showed intrinsic parallelism of some comparison problems such as finding the maximum in a list, sorting and merging of two sorted sets of size $N$ and $M$, respectively, using $\sqrt{NM}$ processors. This merging algorithm required $2 \log \log N + O(1)$ comparison steps, this algorithm can then be applied to sort $N$ keys with $N$ processors in $(2 \log N, \log \log N + O(\log N))$ steps.

In this chapter, some merging algorithms of already sorted sets have been considered and studied where the number of sets to be merged is equal to or greater than the number of available processors. The two-way merge and the odd-even merge algorithms are studied and compared from the efficiency point of view and whether all the processors are fully used. The performance analysis of these algorithms is also included.

Merging by a binary search algorithm is introduced and run on the NEPTUNE system. This algorithm was extended as the Jump Searching algorithm which was derived from the sequential Jump searching that was first proposed by (Shneiderman [1978]). An analysis for the theoretical speed-up of these algorithms was carried out through some rules of probability theory.

In the merge algorithms implemented on the NEPTUNE system, a sorting algorithm was required to sort the sets which are to be merged. Two sorting algorithms were implemented; the bubble sort and the neighbour sort where both algorithms presented a considerable speed-up.
6.2 SORTING ALGORITHMS

The idea of the algorithms presented in this Chapter is to sort a set of data of size N in ascending order. One way of doing this in a parallel MIMD environment is to implement this problem as two relevant components; the sorting and the merging algorithms. Each of these parts is carried out in parallel and the second part is always carried out after the execution of the first part. However, the general idea is to generate M subsets to be carried out in parallel where M is equal to or greater than P, the number of processors in use. First, these subsets are sorted in the sorting part and then merged only after all of the M subsets have been sorted in which case the final required sorted set is obtained.

In this section, we describe the parallel implementation of two sorting algorithms. One algorithm is derived from the well-known sequential bubble sort, while the other is derived from a parallel algorithm which was developed for a SIMD-type computer.

6.2.1 The Bubble Sort

The bubble sort is often known as an exchange selection algorithm (Knuth [1973]). To sort a set S of elements a_1, a_2, ..., a_n, the bubble sort works by comparing a_1 and a_2 then interchanging them if they are out of order and doing the same to a_2, a_3 and a_4 and so on. At the end of this pass, the largest value of all a's tends to move to the right of the set S to occupy the position of a_n. Hence, repetitions of this process will move the appropriate values into positions a_{n-1} and a_{n-2}, etc., so that all the values will be in their final positions and hence the set will be sorted.

As far as the complexity of the algorithm is concerned, the bubble sort is not considered a good algorithm for running on a sequential computer.
However, this is related to its total complexity which is of order \( (N^2) \) in both the worst and expected cases. Obviously, this is inefficient when \( N \) is a very large number. Therefore, to run this algorithm on a parallel machine is a worthwhile task.

The implementation of the bubble sort algorithm on the NEPTUNE system was carried out as follows:

The original input set of \( N \) elements is partitioned into \( M \) subsets of size \( (N/M) \) each, so that the first \( (N/M) \) elements are allocated into the first subset and the next \( (N/M) \) elements into the second subset and so on. We assume that \( M \geq P \), where \( P \) is the number of the available processors.

Each subset is sorted independently of other subsets where the sort is carried out in exactly the same manner as in the sequential form which was described above. When \( M > P \), a processor may execute one or more subsets that are kept in a queue, where each \( P \) subsets can be carried out in parallel (i.e. at the same time). The sorting procedure is complete when all the subsets have been sorted corresponding to their own elements but not amongst themselves.

Now, we consider the total complexity of the bubble sort algorithm when it runs on a sequential machine (i.e. one processor) and when it runs on a \( P \) processors' system. However, in the bubble sort, the total comparisons \( C \) and the number of exchanges \( E_x \) measure the total running-time \( t_1 \) of the algorithm. Knuth [1973] analysed these two quantities for the sequential implementation and he found that on average, these quantities are given as,

\[
C = \frac{1}{2}(N^2 - N\ln N - N)
\]

and

\[
E_x = \frac{1}{4}(N^2 - N)
\]

Then,

\[
t_1 = \frac{1}{2}N^2 - \frac{1}{2}N\ln N - \frac{1}{2}N.
\]

This ensures that the bubble sort is of \( O(N^2) \).
In our implementation of the bubble sort, the set to be sorted is partitioned into as many subsets as the path scheduling list can hold in the NEPTUNE system, where the maximum number of parallel paths to be generated was 75 paths (at the time of the experiments). Therefore, if we generate \( M \) paths with \( M \geq p \), then in each subset \( N/M \) elements are sorted.

Now, if all the paths are carried out on one processor, then by applying equation (6.2.1.1), the total complexity for this algorithm becomes:

\[
t_{\text{IS}} = M \left[ \frac{N^2}{M} - \frac{N}{M} \ln \left( \frac{N}{M} \right) - \frac{N}{M} \right]
\]

\[
= \frac{N^2}{M} - \frac{N}{M} \ln \left( \frac{N}{M} \right) - \frac{N}{M}.
\]

(6.2.1.2)

Meanwhile, when this algorithm is run in parallel with \( P \) processors, \( \left\lceil \frac{N}{P} \right\rceil \) paths have to be carried out by each processor. Thus,

\[
t_{\text{PS}} = \frac{M}{P} \left[ \frac{N^2}{M^2} - \frac{N}{M} \ln \left( \frac{N}{M} \right) - \frac{N}{M} \right]
\]

\[
\leq \frac{N^2}{MP} - \frac{N}{P} \ln \left( \frac{N}{M} \right) - \frac{N}{P} + 1.
\]

(6.2.1.3)

Since we are interested to know how much more efficient is the algorithm when applied on a parallel computer, therefore we measure the speed-up ratio \( S_{\text{PS}}(M) \) for the bubble sort with \( M \) subsets. Thus,

\[
S_{\text{PS}}(M) = \frac{t_{\text{IS}}}{t_{\text{PS}}}
\]

\[
= \frac{1}{t_{\text{PS}}}
\]

\[
= p \left[ 1 - \frac{1}{t_{\text{PS}}} \right].
\]

(6.2.1.4)

which is of \( O(p) \). This means that the optimum linear speed-up is achieved for this implementation. The experimental results of the bubble sort will be included in the sections (6.3) and (6.4) that follows.

6.2.2 The Neighbour Sort

The neighbour sort was first introduced by Haberman [1973] where \( P \) elements are distributed onto \( P \) processors in such a way that there is only one element in each of the processor's memory. This algorithm has been
extended by Baudet and Stevenson [1978] where their algorithm was proposed for a SIMD-type machine. The set of N elements to be sorted is by assigning subsets of elements (i.e. subsets of N) into each processor and each processor is left to sort its own subset. We extended this idea to sort a set of N elements on the NEPTUNE system.

The idea of the algorithm is to partition the set of size N into M subsets of size \( \binom{N}{M} \) each. Each subset is sorted independently of other subsets using the natural two-way merge algorithm (Knuth [1973], pp.161). In other words, each element is compared with its immediate neighbouring element. This results in subsets \( S_i \)'s of two sorted elements within the original subset. In the next stage of the algorithm, each subset \( S_i \) of 2 elements is compared with its immediate neighbouring subset to form a subset of 4 elements. The algorithm then proceeds until the final step where only two subsets within the original subset have to be compared and merged to form the final sorted subset which is of size \( \frac{N}{M} \). The diagram of this algorithm is similar to that in Figure 6.1 but the subsets (1 to 8) are replaced by elements of subsets (i.e. elements from 1 to 8).

To measure the speed-up of this algorithm, it is required to find the total complexity when this sorting algorithm runs in one processor and when it runs in P processors. The complexity of the sequential performance is obtained as follows:

Altogether M subsets are to be sorted, where each subset is of size \( \frac{N}{M} \). By the neighbour sort described above, each subset requires \( \log\left(\frac{N}{M}\right) \) steps. Since the sort procedure is performed by the two-way merge algorithm, the minimum number of comparisons \( c_m \) required to merge two sets of size \( m \) each is given by \( (2m-1)\log m + \log_4 (2m-1) \) (Knuth, [1973], pp.199). Since there is not much difference between the upper and lower bounds of \( c_m \) we choose \( c_m = 2m-1 \). Thus, \( \log\left(\frac{N}{M}\right) \) steps of the sort algorithm are
described as follows:

In the first step, \( \frac{N}{M} \) sequential merges are performed. Thus, in this step where each two neighbours subsets of size one element require at least one comparison. Hence, in the first step, \( \frac{N}{M} \) total comparisons are required. In the second step, \( \frac{N}{M} \) sequential merges are required to sort the subsets of size 2 elements each which are the results of the first step. Then, \( \frac{N}{M} \) comparisons are required. This continues until the final step where only one merge (i.e. \( \frac{1}{2\log(M)} \) merge) is completed to merge the final two subsets of size \( 2^{\log(M)} - 1 \) each. Hence, the total number of comparisons for this step are,

\[
\frac{1}{2\log(M)} - 1 \cdot \frac{N}{M} = \frac{\log(M)}{2} - 1 \cdot \frac{N}{M}.
\]

The sum of the comparisons of the \( \log(M) \) steps gives the total complexity of one subset of size \( \frac{N}{M} \). Thus,

\[
C_s = \frac{N}{M} + \frac{N}{M} + \ldots + \frac{\log(M)}{2\log(M)} - 1 \cdot \frac{N}{M}
\]

\[
= \frac{N}{M} \left(1 - \frac{1}{2^{\log(M)}} + \ldots + 1 - \frac{1}{2^{\log(M)}}\right)
\]

\[
= \frac{N}{M} \left(\log(M) - 1 + \frac{1}{2^{\log(M)}}\right) \text{ comparisons.}
\]

Therefore, for \( M \) subsets

\[
C_{s1} = M \cdot \frac{N}{M} \left(\log(M) - 1 + \frac{N}{M}\right)
\]

\[
= N(\log(M) - 1) + M \text{ comparisons. (6.2.2.1)}
\]

When the algorithm runs on a \( P \) processor's machine, the following complexity is required:

The \( M \) subsets can be sorted in parallel where \( M \) parallel paths are generated with the condition that \( M > P \). In this case, at most \( \left\lfloor \frac{M}{P} \right\rfloor \) parallel paths are performed by each processor where each path requires the comparisons described in the sequential implementation. Therefore, the
total complexity for the sort algorithm in the parallel implementation is

\[ C_{PS} = \frac{N}{P} (\log \frac{N}{M} - 1) + \frac{M}{P} \text{ comparisons.} \]  

(6.2.2.2)

This gives a linear speed-up which is always of \( O(P) \) and does not depend on \( M \).

We conclude that the required time to sort a set of size \( N \) by the neighbour sort algorithm is much less than that of the bubble sort. This is because, the time complexity of the neighbour sort is \( O(N \log \left( \frac{N}{M} \right) \) ), whereas the time complexity of the bubble sort is of \( O(N^2) \) which is not efficient for large \( N \) as far as sequential implementation is concerned.
6.3 MERGING ALGORITHMS

The strategy of merging two disjoint linearly ordered subsets $A$ and $B$ is to determine the linear ordering of their union by means of a sequence of pairwise comparisons between an element of $A$ and an element of $B$.

In this section, we present many merging algorithms applied on subsets which are sorted by one of the sorting algorithms described in Section (6.2).

6.3.1 The Parallel 2-Way Merge Algorithm

The basic principles of the 2-way merge algorithm are given by Knuth [1973, pp.159] as in the following procedure:

Two ordered subsets $a_1 \leq a_2 \leq \cdots \leq a_m$ and $b_1 \leq b_2 \leq \cdots \leq b_n$ are to be merged into a single set $x_1 \leq x_2 \leq \cdots \leq x_{m+n}$. Then,

1. Set $i+1, j+1, k+1$.
2. If $a_i \leq b_j$ go to step (3), otherwise go to (5).
3. Set $x_k = a_i$, $k+k+1$, $i+i+1$, if $i=m$, go to step (2).
4. Set $(x_1, \ldots, x_{m+n})+(b_j, \ldots, b_n)$ and terminate the algorithm.
5. Set $x_k = b_j$, $k+k+1$, $j+j+1$, if $j=n$, go to step (2).
6. Set $(x_1, \ldots, x_{m+n})+(a_i, \ldots, a_m)$ and terminate the algorithm.

In our implementation of the 2-way merge algorithm, we apply the algorithm above on $M$ sorted subsets of size $\frac{N}{M}$ each. We assume that $N$ is divisible by $M$ and $M$ is a power of 2. This merge can be completed in log $M$ steps where parallelism is introduced within each step and not amongst the steps as it is shown in Figure 6.1.

![Figure 6.1: The parallel 2-way merge algorithm](image)
In Figure 6.1, each step can be performed in parallel, where each two neighbouring subsets are merged by one process (or path) to form a subset of size \( \frac{2N}{M} \). Figure 6.1, also illustrates that the number of subsets to be merged is halved in each successive step until the final step where only two subsets are to be merged in which case only one process is required.

In the experiments, the 2-way merge algorithm is implemented to merge the subsets that are sorted by the bubble sort (see Program A.7 in Appendix A) and the neighbour sort. Let us call the two implementations as Prog. 6.1 and Prog. 6.2, respectively.

Now, consider the complexity of the 2-way merge algorithm. We mentioned in Section (6.2.2) that this algorithm requires at least \( \frac{2N}{M} - 1 \) comparisons to merge two subsets of size \( \frac{N}{M} \) each if the data is randomly distributed. Then, we consider first the complexity of the algorithm when it runs in one processor (i.e. sequential implementation).

The algorithm requires \( \log M \) steps. Hence, in the first step, \( \frac{M}{2} \) paths are generated to merge the \( M \) subsets where each path merges two neighbouring subsets to form a subset of size \( \frac{2N}{M} \). Since each path requires \( \frac{2N}{M} - 1 \) comparisons, then the total comparisons required in the first step is \( \frac{M}{2} \left( \frac{2N}{M} - 1 \right) \) comparisons.

Similarly, in the second step, each two subsets of size \( \frac{2N}{M} \) are merged in which case \( \frac{M}{4} \) paths are required. This yields that \( \frac{M}{4} \left( \frac{4N}{M} - 1 \right) \) comparisons are required in this step. The algorithm proceeds until the final step which is the \( \log M \) step. In this step \( \frac{M}{2^{\log M}} = 1 \) path is generated to merge two subsets of size \( \frac{2 \log (M) - 1}{M} \) which gives \( \frac{M}{2^{\log M}} \left( \frac{2 \log (M) - 1}{M} \cdot N - 1 \right) \) comparisons as the complexity of this step. Therefore, the total complexity of the 2-way merge algorithm in one processor is,
If we consider the implementation with the bubble sort (i.e. Prog 6.1), we can obtain the total complexity which represents both the sorting and merging complexities. Since the complexity of the bubble sort on one processor is given by $t_{1S}$ in equation (6.2.1.2), therefore, the total complexity $T_1$ of Prog 6.1 in one processor is given by:

$$T_1 = t_{1S} + t_{1M} = \frac{N^2}{M} - 2N\log \left(\frac{N}{M}\right) - \frac{3N}{M} + N\log M - M + 1 \quad (6.3.1.2)$$

For the parallel implementation of the 2-way merge, we proceed with the following analysis:

Since $\log M$ steps are sufficient to merge the $M$ sorted subsets to the final sorted list, and the number of paths is halved in each step, therefore, if $M > P$, then each processor has to carry out one or more paths in the first few steps. Precisely, when, $M \geq 2^iP$, for $i=1,2,\ldots$, all the co-operating processors are potentially activated so that no loss in the algorithm efficiency is achieved. In other words, when the step number $i$ such that $i \leq \log \left(\frac{M}{P}\right)$, for $i=1,2,\ldots$, all the processors are potentially used. After these $\log \left(\frac{M}{P}\right)$ steps, the number of the processors is halved in each step until the final step where only one processor is active while $(P-1)$ processors remain idle. The number of steps between $\log \left(\frac{M}{P}\right)$ and the final step are $\log P$ steps, where $\log \left(\frac{M}{P}\right) + \log P = \log M$ steps represent the total number of steps to merge the $M$ subsets.

$$t_{1M} = \frac{M}{2} \left(\frac{2N}{M} - 1\right) + \frac{M}{4} \left(\frac{4N}{M} - 1\right) + \cdots + \frac{M}{2^{\log M}} \left(\frac{2^{\log M} N}{M} - 1\right)$$

$$= N - \frac{M}{2} + N - \frac{M}{4} + \cdots + N - \frac{M}{2^{\log M}}$$

$$= N \log M - \frac{M}{2}.2(1 - \frac{1}{2^{\log M}})$$

$$= N \log M - M + 1$$
The formula for the total comparisons in the parallel form of the merge algorithm will then be as follows:

In the first step, \( \left\lceil \frac{M}{2P} \right\rceil \) paths are to be carried out by each processor resulting in \( \frac{M}{2P} \left( \frac{2N}{M} - 1 \right) \) comparisons as the complexity for each processor. In the second step, however, \( \frac{M}{4P} \left( \frac{4N}{M} - 1 \right) \) comparisons are performed by each processor, and so on until the \( \log_2 \left( \frac{M}{P} \right) \) step where \( \frac{M}{2^{\log_2 \left( \frac{M}{P} \right)} . P} \left( \frac{2 \log \left( \frac{M}{P} \right)}{M} \right) . N - 1 \) comparisons are required.

In the \( \left( \log_2 \left( \frac{M}{P} \right) + 1 \right) \)th step, where the number of processors \( P \) is halved, \( \frac{M}{2^{\log_2 \left( \frac{M}{P} \right)} + 1} \cdot \frac{N}{P} \) comparisons are performed. Similarly, in the \( \left( \log_2 \left( \frac{M}{P} \right) + 2 \right) \)th step, \( \frac{M}{2^{\log_2 \left( \frac{M}{P} \right)} + 2} \cdot \frac{N}{P} \) comparisons are required where \( P \) is divided by \( \frac{P}{4} \) in this case. This continues in a similar manner until the final step is reached, i.e. the \( \left( \log_2 \left( \frac{M}{P} \right) + \log_2 \left( \frac{M}{P} \right) \right) \) step, in which case only one processor remains active and the number of comparisons required are \( \frac{M}{2^{\log_2 \left( \frac{M}{P} \right)} \cdot \frac{P}{2^{\log_2 \left( \frac{M}{P} \right)}}} \left( \frac{2 \log \left( \frac{M}{P} \right)}{M} \right) . N - 1 \). Summing up the comparisons in all the steps, we obtain the total complexity \( t_{PM} \) for the 2-way merge when run on \( P \) processors. Thus,

\[
t_{PM} = \frac{M}{2P} \left( \frac{2N}{M} - 1 \right) + \frac{M}{4P} \left( \frac{4N}{M} - 1 \right) + \ldots + \frac{M}{2^{\log_2 \left( \frac{M}{P} \right)} \cdot P} \left( \frac{2 \log \left( \frac{M}{P} \right)}{M} . N - 1 \right)
\]

\[
+ \frac{M}{2^{\log_2 \left( \frac{M}{P} \right)} + 1} \cdot \frac{N}{P} + \frac{M}{2^{\log_2 \left( \frac{M}{P} \right)} + 2} \cdot \frac{N}{P} + \ldots + \frac{M}{2^{\log_2 \left( \frac{M}{P} \right)} + \log_2 \left( \frac{M}{P} \right)} \cdot \frac{N}{P} + \frac{M}{2^{\log_2 \left( \frac{M}{P} \right)} + \log_2 \left( \frac{M}{P} \right) + 1} \cdot \frac{N}{P}
\]

\[
\leq \frac{N}{P} - \frac{M}{2P} + 1 + \frac{N}{4P} - \frac{M}{4P} + 1 + \ldots + \frac{N}{P} - \frac{M}{2^{\log_2 \left( \frac{M}{P} \right)} \cdot P} + 1 + \frac{2N}{P} - \frac{M}{2^{\log_2 \left( \frac{M}{P} \right)} \cdot P} + 1 + \frac{2N}{P} - \frac{M}{2^{\log_2 \left( \frac{M}{P} \right)} \cdot P} + 1 + \ldots + \frac{M}{2^{\log_2 \left( \frac{M}{P} \right)} \cdot P} + 1
\]

\[
\leq 2N + \frac{M}{P} + 1 \quad (6.3.1.3)
\]

Inequality (6.3.1.3) can be simplified with the use of the rules of the geometric series \( 1 + r + r^2 + r^3 + \ldots + r^{n-1} \) whose sum equals,
Thus, inequality (6.3.1.3) becomes
\[
\frac{n\frac{n - 1}{r - 1}}{\frac{1}{r - 1} \frac{r - n}{1 - r}} \text{ if } r > 1, \text{ and } \frac{1 - r^n}{1 - r} \text{ if } r < 1.
\]

Therefore, inequality (6.3.1.3) becomes
\[
\begin{align*}
t_{PM} & \leq \frac{N}{P} \log \left( \frac{M}{P} \right) - \frac{M}{2P}(1 + 1 + \ldots + \frac{1}{2^\log \left( \frac{M}{P} \right) - 1}) + \log \left( \frac{M}{P} \right) + 2N(1 + 2 + \ldots + 2^{\log \left( \frac{P}{M} \right) - 1}) \\
& \quad - \frac{M}{P} \log P + \log P \\
& \quad \leq \frac{N}{P} \log \left( \frac{M}{P} \right) - \frac{M}{P} \left( 1 - \frac{1}{2^\log \left( \frac{M}{P} \right) - 1} \right) + \log \left( \frac{M}{P} \right) + 2N\left( 2^{\log \left( \frac{P}{M} \right) - 1} \right) \\
& \quad \leq \frac{N}{P} \log \left( \frac{M}{P} \right) + 2N\left( \frac{P}{M} - 1 \right) - \frac{M}{P} + 1 + \log \left( \frac{M}{P} \right). \\
\end{align*}
\]

(6.3.1.4)

If we sum up this result with the complexity of the bubble sort when run on P processors which is given by equation (6.2.1.3), we obtain the total complexity \( T \) of Prog 6.1 when P processors are used. Thus,
\[
\begin{align*}
T_P & = t_{PS} + t_{PM} \\
& \leq \frac{N^2}{MP} - \frac{N}{P} \log \left( \frac{N}{M} \right) - \frac{N}{P} \log \left( \frac{M}{P} \right) + N \log M - M + 1 \\
& \quad + \frac{2N}{P} \log \left( \frac{M}{P} \right) - M + 1 + \log \frac{M}{P} \\
& \quad \leq \frac{P \log M}{\log \left( \frac{M}{P} \right) + 2P - 2} \left[ 1 - \frac{\frac{M}{P} - 1 - \log \left( \frac{M}{P} \right)}{t_{PM}} \right]. \\
\end{align*}
\]

(6.3.1.5)

The speed-up ratios for the merge algorithm alone \( S_{merge(M)} \) and the total speed-up for both the sort and the merge algorithms together \( S_{total(M)} \) can now be represented.
\[
\begin{align*}
S_{merge(M)} &= \frac{t_{PS}}{t_{PM}} \\
& \geq \frac{N \log M - M + 1}{\frac{N}{P} \log \left( \frac{M}{P} \right) + \frac{2N}{P}(P - 1) - \frac{M}{P} + 1 + \log \frac{M}{P}} \\
& \geq \frac{P \log M}{\log \left( \frac{M}{P} \right) + 2P - 2} \left[ 1 - \frac{\frac{M}{P} - 1 - \log \left( \frac{M}{P} \right)}{t_{PM}} \right]. \\
\end{align*}
\]

(6.3.1.6)

Thus, the \( S_{merge(M)} \) is of \( O\left( \frac{P \log M}{\log \left( \frac{M}{P} \right) + 2P - 2} \right) \) which is approximately what was obtained experimentally. Meanwhile, the total speed-up is measured as:
\[
\begin{align*}
S_{total(M)} &= \frac{T_1}{T_{PS}} \\
& \geq \frac{\frac{N}{P^2} \left( \frac{N}{M} - 1 \right) \log \left( \frac{N}{M} \right) - \frac{3N}{4P} + N \log M - M + 1}{\frac{1}{M} \left[ N^2 - 1 \right] - \frac{N}{P^2} \log \left( \frac{N}{M} \right) - \frac{3N}{4P} + N \log \left( \frac{M}{P} \right) + \frac{2N}{P}(P - 1) - \frac{M}{P} + 1 + \log \left( \frac{M}{P} \right) + 2} \\
& \geq \frac{P \log M}{\log \left( \frac{M}{P} \right) + 2P - 2} \left[ 1 - \frac{\frac{M}{P} - 1 - \log \left( \frac{M}{P} \right)}{t_{PM}} \right]. \\
\end{align*}
\]
\[
N(2P - \log P + \frac{17}{4}) + P \log \left(\frac{M}{P}\right) + 2P-1
\geq P - \frac{T_P}{P}.
\] (6.3.1.7)

It can be seen that as \( M \) increases \( T_P \) decreases and \( S_{\text{total}} \) slightly decreases away from \( P \). Whereas, as \( M \) increases \( S_{\text{merge}} \) increases as it is always of \( O\left(\frac{P \log M}{\log \left(\frac{M}{P}\right) + 2P-2}\right) \).

From inequalities (6.3.1.6) and (6.3.1.7) above we can represent the theoretical values of the total speed-up and the merge speed-up which in actual fact are greater than the experimental values, since we included in the analysis only the total number of comparisons and exchanges that dominate the performance of the algorithm and did not include any data communication or parallel allocation overheads that might degrade the parallelism. The results obtained from the NEPTUNE system of Prog 6.1 with \( N=1024 \) elements are listed in Table 6.1.

From the formulae given in (6.3.1.6) and (6.3.1.7) we measure the theoretical speed-up and tabulate these results together with the experimental results in Table 6.2.
<table>
<thead>
<tr>
<th>No. of processors (P)</th>
<th>No. of paths (M)</th>
<th>Time for sorting (sec)</th>
<th>Time for merging (sec)</th>
<th>Total time (sec)</th>
<th>Speed-up for sorting</th>
<th>Speed-up for merging</th>
<th>Speed-up for both parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>88.050</td>
<td>1.570</td>
<td>89.620</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>47.630</td>
<td>1.230</td>
<td>48.860</td>
<td>1.85</td>
<td>1.28</td>
<td>1.83</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>23.330</td>
<td>1.220</td>
<td>24.55</td>
<td>3.77</td>
<td>1.29</td>
<td>3.65</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>43.080</td>
<td>2.340</td>
<td>45.420</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>23.270</td>
<td>1.620</td>
<td>24.890</td>
<td>1.85</td>
<td>1.44</td>
<td>1.825</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>16.910</td>
<td>1.600</td>
<td>18.510</td>
<td>2.55</td>
<td>1.46</td>
<td>2.45</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>11.640</td>
<td>1.450</td>
<td>13.090</td>
<td>3.70</td>
<td>1.60</td>
<td>3.47</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>20.970</td>
<td>3.110</td>
<td>24.080</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>11.260</td>
<td>2.070</td>
<td>13.330</td>
<td>1.86</td>
<td>1.52</td>
<td>1.81</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>8.680</td>
<td>1.920</td>
<td>10.600</td>
<td>2.41</td>
<td>1.62</td>
<td>2.27</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>5.890</td>
<td>1.680</td>
<td>7.570</td>
<td>3.56</td>
<td>1.85</td>
<td>3.18</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>10.250</td>
<td>3.880</td>
<td>14.140</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>5.670</td>
<td>2.450</td>
<td>8.120</td>
<td>1.81</td>
<td>1.58</td>
<td>1.74</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>4.220</td>
<td>2.210</td>
<td>6.430</td>
<td>2.43</td>
<td>1.76</td>
<td>2.20</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>2.950</td>
<td>1.080</td>
<td>4.030</td>
<td>3.47</td>
<td>2.10</td>
<td>2.93</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
<td>4.850</td>
<td>4.700</td>
<td>9.550</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>2.660</td>
<td>2.940</td>
<td>5.600</td>
<td>1.82</td>
<td>1.60</td>
<td>1.70</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>1.990</td>
<td>2.520</td>
<td>4.510</td>
<td>2.45</td>
<td>1.87</td>
<td>2.12</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>1.430</td>
<td>2.120</td>
<td>3.550</td>
<td>3.39</td>
<td>2.22</td>
<td>2.69</td>
</tr>
</tbody>
</table>

**TABLE 6.1:** The bubble sort algorithm with the 2-way merge algorithm for data size (1024)
<table>
<thead>
<tr>
<th>Data (N)</th>
<th>Processors (P)</th>
<th>Paths (M)</th>
<th>( S_{merge} )</th>
<th>( S_{total} )</th>
<th>Efficiency (EP)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Theory</td>
<td>Experiments</td>
<td>Theory</td>
</tr>
<tr>
<td>(1024)</td>
<td>2</td>
<td>4</td>
<td>1.33</td>
<td>1.28</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8</td>
<td>1.50</td>
<td>1.44</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16</td>
<td>1.60</td>
<td>1.52</td>
<td>0.80</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>32</td>
<td>1.66</td>
<td>1.58</td>
<td>0.83</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>64</td>
<td>1.71</td>
<td>1.60</td>
<td>0.86</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8</td>
<td>1.66</td>
<td>1.46</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>16</td>
<td>1.86</td>
<td>1.62</td>
<td>0.62</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>32</td>
<td>2.01</td>
<td>1.76</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>64</td>
<td>2.13</td>
<td>1.87</td>
<td>0.71</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>1.33</td>
<td>1.29</td>
<td>0.33</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>1.71</td>
<td>1.60</td>
<td>0.43</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>1.99</td>
<td>1.85</td>
<td>0.50</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>32</td>
<td>2.21</td>
<td>2.10</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>64</td>
<td>2.39</td>
<td>2.22</td>
<td>0.60</td>
</tr>
</tbody>
</table>

**TABLE 6.2:** The theoretical and experimental values of the merge speed-up and the total speed-up of Prog 6.1.
The efficiency $E_P$ in Table 6.2 is obtained theoretically from the formula,

$$E_P = \frac{\text{Speed-up}}{P}$$

The theoretical and experimental values of the speed-up in Table 6.2 are represented graphically in Figure 6.2 where the number of processors $P$ is 4.

From the above analysis, it is obvious that the total speed-up is $O(P)$. In Figure 6.3, we notice that the maximum total speed-up occurs when $M=2P$ where the speed-up is slightly less than the "linear speed-up". This difference, in fact, is due to data communication, parallel path allocation and shared data requirements. It can also be seen that the speed-up decreases as $M$ increases. This is due to the synchronisations involved at the end of the parallel paths when $M>P$.

On the other hand, the merge speed-up in Figure 6.3 increases as $M$ increases and the maximum speed-up occurs at $M=64$ (the maximum number of paths we could generate on the NEPTUNE system, at the time of the experiments, where $M$ is a power of 2). This increase is due to the nature of the algorithm that halves the number of paths in each step. This means that in the first $\log_2 \left(\frac{M}{P}\right)$ steps all the processors are active and contribute to the solution of the problem and after $\log_2 \left(\frac{M}{P}\right)$ steps the number of processors is halved until the final step is reached where only one processor is active while the other processors remain idle. Therefore, if $M>>P$, then $\log_2 \left(\frac{M}{P}\right)$ is large hence all the processors are significantly active. Although, the timing results increases as $M$ increases in the merge algorithm the speed-up results are improved for large $M$.

Now, in order to derive a relation between $M$ and $P$ for a particular data size $N$, we give different values for $M$ and $P$ to the $S_{\text{total}}$ in inequality (6.3.1.7) which measures the theoretical total speed-up. These results are listed in Table 6.3.
The Theoretical and Experimental Speed-up Results of Prog 6.1
FIGURE 6.3
The Experimental Total and Merge Speed-up Values of Prog 6.1
For the best decomposition for sorting the original set we can derive from the results the following formula that relates $M$ to $P$ and $N$.

Thus,

$$\log_2 N \leq M \leq P \cdot \left(\frac{\log N}{2}\right) - 1.$$  \hspace{1cm} (6.3.1.8)

<table>
<thead>
<tr>
<th>$N$</th>
<th>$M$</th>
<th>$P$</th>
<th>$S_{\text{total}}$</th>
<th>$N$</th>
<th>$M$</th>
<th>$P$</th>
<th>$S_{\text{total}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1024)</td>
<td>8</td>
<td>2</td>
<td>1.98</td>
<td>(2048)</td>
<td>8</td>
<td>2</td>
<td>1.99</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2.93</td>
<td></td>
<td></td>
<td>3</td>
<td>2.96</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3.84</td>
<td></td>
<td></td>
<td>4</td>
<td>3.92</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4.72</td>
<td></td>
<td></td>
<td>5</td>
<td>4.83</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5.57</td>
<td></td>
<td></td>
<td>6</td>
<td>5.77</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>6.39</td>
<td></td>
<td></td>
<td>7</td>
<td>6.68</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7.17</td>
<td></td>
<td></td>
<td>8</td>
<td>7.56</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>7.94</td>
<td></td>
<td></td>
<td>9</td>
<td>8.43</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>1.96</td>
<td></td>
<td>10</td>
<td>2</td>
<td>1.98</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2.86</td>
<td></td>
<td>11</td>
<td>3</td>
<td>2.93</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3.70</td>
<td></td>
<td>12</td>
<td>4</td>
<td>3.84</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4.48</td>
<td></td>
<td>13</td>
<td>5</td>
<td>4.72</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5.21</td>
<td></td>
<td></td>
<td>6</td>
<td>5.57</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>5.31</td>
<td></td>
<td></td>
<td>7</td>
<td>6.39</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>6.53</td>
<td></td>
<td></td>
<td>8</td>
<td>7.65</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>7.13</td>
<td></td>
<td></td>
<td>9</td>
<td>8.68</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>1.93</td>
<td></td>
<td>10</td>
<td>2</td>
<td>1.96</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2.75</td>
<td></td>
<td>11</td>
<td>3</td>
<td>2.77</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3.47</td>
<td></td>
<td>12</td>
<td>4</td>
<td>3.70</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4.10</td>
<td></td>
<td>13</td>
<td>5</td>
<td>4.19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>4.68</td>
<td></td>
<td></td>
<td>6</td>
<td>4.82</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>5.19</td>
<td></td>
<td></td>
<td>7</td>
<td>5.41</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>5.65</td>
<td></td>
<td></td>
<td>8</td>
<td>5.93</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>6.05</td>
<td></td>
<td></td>
<td>9</td>
<td>8.68</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 6.3: Theoretical total speed-up of Prog 6.1 that are derived from inequality (6.3.1.7)
In other words, for fixed values of \( P \) and \( N \) we obtain the corresponding maximum value of \( M \). However, from Table 6.3, we can obtain the maximum number of processors for the speed-up to remain linear (i.e. of \( O(P) \)), when a particular number of paths is used for a given data size. This number of processors is given by:

\[
P \leq \log\left(\frac{N}{M}\right) + 1. 
\]  

(6.3.1.9)

However, by examining Table 6.3, we notice that the speed-up is approximately linear for all \( P \leq \log\left(\frac{N}{M}\right) + 1 \).

So far, a discussion of the 2-way merge algorithm when implemented with the Bubble sort algorithm was presented. Now, we describe the complexity measurements of Prog 6.2 which implements the 2-way merge with the neighbour sort algorithm. First, consider the complexity \( T_1 \) of Prog. 6.2 when run sequentially (i.e. in one processor). This complexity is the sum of the sorting complexity given by equation (6.2.2.1) and the merging complexity given by equation (6.3.1.1). Thus,

\[
T_1 = C_{1S} + t_{1M}
\]

\[
= N(\log\left(\frac{N}{M}\right) - 1) + M + N\log M - M + 1
\]

\[
= N(\log N - 1) + 1. 
\]  

(6.3.1.10)

In the parallel implementation, the total complexity \( T_p \) is given by the sum of the sorting complexity and the merging complexity when they are run in parallel and given by equations (6.2.2.2) and (6.3.1.4), respectively. Thus,

\[
T_p = \frac{N}{P} \left(\log\left(\frac{N}{P}\right) - 1\right) + \frac{M}{P} + \frac{N}{P} \log\left(\frac{M}{P}\right) + \frac{2N}{P} (P-1) - \frac{M}{P} + 2 + \log\left(\frac{M}{P}\right)
\]

\[
= \frac{N}{P} \left(\log\left(\frac{N}{P}\right) - 1\right) + \frac{2N}{P} (P-1) + \log\left(\frac{M}{P}\right) + 2. 
\]  

(6.3.1.11)

Then, the total speed-up can be obtained which is,

\[
S_{\text{total}} = \frac{N(\log N - 1) + 1}{\frac{N}{P} \left(\log\left(\frac{N}{P}\right) - 1\right) + \frac{2N}{P} (P-1) + \log\left(\frac{M}{P}\right) + 2} 
\]  

(6.3.1.12)
From equation (6.3.1.12), we notice that the speed-up is mainly dependent upon the number of processors $P$ and independent of the number of paths $M$. This means that for a particular value of $P$, the speed-up values are the same for any number of paths $M$. When equation (6.3.1.12) is simplified in a manner similar to before, $S_{total}$ becomes of

$$O\left(\frac{P(\log N-1)}{\log (\frac{N}{P})+2P-2}\right),$$

whilst the total speed-up of Prog 6.1 was of $O(P)$. Although, the former speed-up is less than the latter, the required sorting time in the neighbour sort is much less than that of the bubble sort. This is because, the time complexity of the neighbour sort is $O(N(\log(\frac{N}{M})-1))$ whereas in the bubble sort it is of $O(N^2/M)$. This degradation of the speed-up is clear from the experimental results obtained from the NEPTUNE system which are listed in Table 6.4. The theoretical speed-up values presented in Table 6.5 fairly match the experimental values of the speed-up.
<table>
<thead>
<tr>
<th>No.of processors</th>
<th>No.of paths</th>
<th>Time for sorting</th>
<th>Time for merging</th>
<th>Total time</th>
<th>Speed-up for sorting</th>
<th>Speed-up for merging</th>
<th>Speed-up for both parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>5.360</td>
<td>1.570</td>
<td>6.930</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2.890</td>
<td>1.230</td>
<td>4.120</td>
<td>1.85</td>
<td>1.28</td>
<td>1.68</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1.470</td>
<td>1.230</td>
<td>2.700</td>
<td>3.65</td>
<td>1.28</td>
<td>2.57</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>4.600</td>
<td>2.340</td>
<td>6.940</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2.490</td>
<td>1.650</td>
<td>4.140</td>
<td>1.85</td>
<td>1.42</td>
<td>1.68</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>1.830</td>
<td>1.600</td>
<td>3.430</td>
<td>2.51</td>
<td>1.46</td>
<td>2.02</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>1.290</td>
<td>1.450</td>
<td>2.740</td>
<td>3.57</td>
<td>1.60</td>
<td>2.53</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>3.860</td>
<td>3.110</td>
<td>6.970</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>2.100</td>
<td>2.070</td>
<td>4.170</td>
<td>1.84</td>
<td>1.52</td>
<td>1.67</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>1.470</td>
<td>1.920</td>
<td>3.380</td>
<td>2.63</td>
<td>1.62</td>
<td>2.06</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>1.080</td>
<td>1.680</td>
<td>2.730</td>
<td>3.57</td>
<td>1.85</td>
<td>2.55</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>3.150</td>
<td>3.880</td>
<td>7.040</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>1.690</td>
<td>2.450</td>
<td>4.190</td>
<td>1.86</td>
<td>1.58</td>
<td>1.68</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>1.170</td>
<td>2.210</td>
<td>3.400</td>
<td>2.69</td>
<td>1.76</td>
<td>2.07</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>0.890</td>
<td>1.880</td>
<td>2.780</td>
<td>3.54</td>
<td>2.10</td>
<td>2.53</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
<td>2.550</td>
<td>4.700</td>
<td>7.240</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>1.380</td>
<td>2.940</td>
<td>4.320</td>
<td>1.85</td>
<td>1.60</td>
<td>1.68</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>0.960</td>
<td>2.520</td>
<td>3.480</td>
<td>2.66</td>
<td>1.87</td>
<td>2.08</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>0.750</td>
<td>2.120</td>
<td>2.880</td>
<td>3.40</td>
<td>2.22</td>
<td>2.51</td>
</tr>
</tbody>
</table>

**TABLE 6.4:** The experimental results of Prog 6.2: the neighbour sort + the 2-way merge algorithm for N=1024
6.3.2 The Parallel Odd-Even Merge Algorithm

The odd-even merge algorithm is based on Batcher's O-1 merge (Batcher [1968]) which was improved by Baudet and Stevenson [1978] to make it more suitable for a SIMD-type computer. We extend this latter version so that it can be applied on a MIMD-type computer. This algorithm is described as follows:

The original set of \( N \) elements is partitioned into \( M \) subsets of size \( \left( \frac{N}{M} \right) \) each. These subsets can be merged by the odd-even merge algorithm in at most \( M \) sequential steps (see Baudet and Stevenson [1978]) where the parallelism is introduced within each step similar to the 2-way merge. The graphical representation of this merge algorithm is shown in Figure 6.4 for \( M=8 \).

In the steps 1, 3, 5, ..., \( M-1 \), \( s_i \) is merged with \( s_{i+1} \) for \( i=1, 3, 5, ..., M-1 \),
where \( M \) is assumed to be a power of 2. An auxiliary list \( C \) is used to hold the partial results of the merge. When \( S_i \) and \( S_{i+1} \) have been completely merged the results that are stored in \( C \) are assigned back to the original list \( A \).

Similarly, in steps \( i \), for \( i=2,4,6,\ldots,M-2 \), \( S_i \) and \( S_{i+1} \) are merged in a similar process as mentioned above. Generally speaking in the odd numbered steps, each of the odd numbered subsets are merged with its neighbour subset which is even numbered, and vice versa for the even numbered subsets. Each two subsets are merged using the sequential 2-way merge to form one subset of size equal to the sum of the two subsets. In the next step of the algorithm, the appropriate half of the resultant subset is merged with the neighbouring half of the next subset and so on as shown in Figure 6.4.

In the experiments, the odd-even merge algorithm was implemented with the Bubble sort and the Neighbour sort algorithm which sorts the \( M \) subsets.
which are ready to be merged. The two programs of these implementations are called Prog 6.3 and Prog 6.4 respectively and we choose, for illustration, the program for the odd-even merge with the neighbour sort which is listed in Program A.8 of Appendix A.

Now, for the complexity of the odd-even merge algorithm on one processor we proceed as follows:

As $M$ steps have to be performed on $M$ sorted subsets in order to form the final sorted set, hence $\frac{M}{2}$ odd steps and $\frac{M}{2}$ even steps have to be completed. In each of the $\frac{M}{2}$ odd steps, $\frac{M}{2}$ paths are generated where each path merges two subsets of size $\frac{N}{M}$ each. As $\left(\frac{2N}{M} - 1\right)$ comparisons are required to merge two subsets of size $\frac{N}{M}$ each, therefore, the total comparisons required in all the odd steps are:

$$C_O = \frac{M}{2} \cdot \frac{M}{2} \left(\frac{2N}{M} - 1\right)$$

$$= \frac{N}{4} M^2 - \frac{M^2}{4} \cdot \frac{2}{2}$$  \hspace{1cm} (6.3.2.1)

Similarly, $\frac{M}{2}$ even steps are performed where in each step, $\left(\frac{M}{2} - 1\right)$ paths are generated to merge the subsets of size $\frac{N}{M}$. This, however, gives the total number of comparisons in the even steps as:

$$C_E = \frac{M}{2} \left(\frac{M}{2} - 1\right) \left(\frac{2N}{M} - 1\right)$$

$$= N\left(\frac{M}{2} - 1\right) - \frac{M^2}{2} \cdot \frac{2}{2}$$  \hspace{1cm} (6.3.2.2)

Thus, the total comparisons $C_{1M}$ of the sequential implementation of the merge algorithm is given by,

$$C_{1M} = C_O + C_E$$

$$= \frac{N}{4} M^2 - \frac{M^2}{4} + N\left(\frac{M}{2} - 1\right) - \frac{M^2}{2} \cdot \frac{2}{2}$$

$$= N(M-1) - \frac{M^2}{4}(M-1)$$  \hspace{1cm} (6.3.2.3)

If we consider Prog 6.3 of the bubble sort and the odd-even merge, we can find the total complexity of the program with the use of the bubble sort complexity of equation (6.2.1.2) which is given by:
In the parallel implementation of this algorithm, three cases have to be considered. First, when \( \frac{M}{2} < P \), \( \frac{M}{2} \) paths are generated in each odd step and \( \left\lfloor \frac{P - 2P}{M} \right\rfloor \) processors out of \( P \) processors are fully used to complete the work. Therefore the complexity for the \( \frac{M}{2} \) odd steps with \( \frac{M}{2} \) paths each is given by:

\[
C_{OP} = \left\lfloor \frac{M}{2} \right\rfloor \cdot \frac{1}{\left\lfloor \frac{P - 2P}{M} \right\rfloor} \cdot \frac{M}{2} \cdot (\frac{2N}{M} - 1).
\]

However, in the \( \frac{M}{2} \) even steps, \( \left\lfloor \frac{P - 2P}{M} \right\rfloor \) paths are generated where only \( \left\lfloor \frac{P - 2P}{M} \right\rfloor \) processors are participating. Therefore, the total complexity in the even steps is given by,

\[
C_{EP} = \left\lfloor \frac{M}{2} \right\rfloor \cdot \frac{1}{\left\lfloor \frac{P - 2P}{M} \right\rfloor} \cdot \frac{M}{2} \cdot (\frac{2N}{M} - 1).
\]

The total complexity of all the steps in this case then becomes:

\[
C_{PM} = \left\lfloor \frac{M}{2} \right\rfloor \cdot \frac{1}{\left\lfloor \frac{P - 2P}{M} \right\rfloor} \cdot \frac{M}{2} \cdot (\frac{2N}{M} - 1) + \left\lfloor \frac{M}{2} \right\rfloor \cdot \frac{1}{\left\lfloor \frac{P - 2P}{M} \right\rfloor} \cdot \frac{M}{2} \cdot (\frac{2N}{M} - 1)
\]

\[
\leq \frac{NM}{2} \cdot \frac{M}{(PM-2P)} - \frac{M^2}{4} \cdot \frac{M}{(PM-2P)} + \frac{N}{2} \cdot \frac{M}{(PM-2P)} - \frac{M}{4} \cdot \frac{N}{(PM-2P)} + \frac{M}{4} \cdot \frac{M}{(PM-2P)} + 2
\]

\[
\leq \frac{NM}{2} \cdot \frac{M-1}{MP-2P} - \frac{M^2}{2P} \cdot \frac{M-1}{MP-2P} + 2.
\]

The second case is when \( \frac{M}{2} = P \). Since in each of the \( \frac{M}{2} \) odd steps, \( \frac{M}{2} \) paths are generated, therefore all the processors become fully used and none of them remain idle. However, each processor carries out \( \left\lfloor \frac{M}{2} \right\rfloor \cdot \frac{1}{P} \) paths and this yields the following complexity for all the odd steps. Thus,

\[
C_{OP} = \left\lfloor \frac{M}{2} \right\rfloor \cdot \frac{1}{P} \cdot \frac{M}{2} \cdot (\frac{2N}{M} - 1).
\]

On the other hand, in the \( \frac{M}{2} \) even steps, \( \left\lfloor \frac{M}{2} \right\rfloor - 1 \) paths are generated hence only \( (P-1) \) processors are required in this case and only one processor
remains idle. Therefore, in the even steps, the total complexity becomes:

\[ C_{EP} = \left( \frac{M}{2} - 1 \right) \left( \frac{1}{P-1} \right) \frac{M}{2} \left( \frac{2N}{M} - 1 \right). \]

Summing up \( C_{OP} \) and \( C_{EP} \) to give the total complexity \( C_{PM} \) as:

\[ C_{PM} = \left[ \frac{M}{2} \cdot \frac{1}{P} \right] \frac{M}{2} \left( \frac{2N}{M} - 1 \right) + \left[ \frac{M}{2} \cdot \frac{1}{(P-1)} \right] \frac{M}{2} \left( \frac{2N}{M} - 1 \right) \]

\[ \leq \frac{1}{M} M \frac{M}{2} N \frac{(M-2)}{2(P-1)} M / 2(M/2 - 1) + 2 \]

\[ \leq N(M - M - 2) - \frac{M}{P} - \frac{M/2(M/2 - 1)}{(P-1)} + 2 \]

\[ \leq N(2P - M - 2) - \frac{M}{P} - \frac{N/2(M/2 - 1)}{(P-1)} + 2 \quad (6.3.2.6) \]

Finally, in the case \( M > P \), all \( P \) processors are active and each processor has an equal share of paths, that is \( \left[ \frac{M}{2} \cdot \frac{1}{P} \right] \) paths in the odd numbered steps and \( \left[ \frac{M}{2} - 1 \right] \frac{1}{(P-1)} \) paths in the even numbered steps. Therefore, the total complexity for the algorithm when \( M > P \) is:

\[ C_{PM} = \left( \frac{M}{2} \cdot \frac{1}{P} \right) \frac{M}{2} \left( \frac{2N}{M} - 1 \right) + \left( \frac{M}{2} - 1 \right) \frac{1}{P} \frac{M}{2} \left( \frac{2N}{M} - 1 \right) \]

\[ \leq N(M-1) - \frac{M^2}{2P} + \frac{N}{2P} + 2. \quad (6.3.2.7) \]

Now, we can measure the merge speed-up, \( S_{merge}(M) \), of \( M \) subsets by dividing \( C_{LM} \) of equation (6.3.2.3) by \( C_{PM} \) of equations (6.3.2.5), (6.3.2.7). Thus,

\[ S_{merge}(M) \geq \begin{cases} \frac{(N(M-1) - \frac{1}{2} (M-M))}{(N(M-1) - \frac{1}{2} (M-M))} / \left( N \left( \frac{2P-M-1}{2P(P-1)} + \frac{M^2}{4P} - \frac{M/2(M/2 - 1)}{P-1} + 2 \right) \right) & \text{for } M = P \\ \frac{(N(M-1) - \frac{1}{2} (M-M))}{(N(M-1) - \frac{1}{2} (M-M))} / \left( N \left( M^2 + \frac{M}{2P} \right) + 2 \right), & \text{for } M > P \end{cases} \quad (6.3.2.8) \]

These speed-up formulae can be simplified as an order quantity, such as:

\[ S_{merge}(M) \geq \begin{cases} O\left( \frac{P(M-2)}{M} \right), & \text{for } M = P \\ O\left( \frac{2P(P-1)(M-1)}{2P-1} \right), & \text{for } M > P \end{cases} \quad (6.3.2.9) \]
From the formulae in (6.3.2.9), it is clear that the merge speed-up increases as \( M \) increases and becomes \( O(P) \) when \( M \geq \frac{P}{2} \) is much greater than \( P \) which is confirmed by the experimental results of Prog 6.3 when this algorithm is implemented with bubble sort. These experimental results are listed in Table 6.6. The total speed-up in Table 6.6 refers to the ratio of the total time of the sort and merge algorithms in one processor to that for \( P \) processors. However, the total speed-up, \( S_{\text{total}} \) can be measured from the above formulae. Thus, the total time complexity \( T_P \) is measured, where the sort complexity of equation (6.2.1.3) is used to give,

\[
T_P = t_{PS} + C_{PM}.
\]

The total speed-up, \( S_{\text{total}} \), is measured as follows:

\[
S_{\text{total}} = \left\{ \begin{array}{ll}
\frac{N^2}{MP} - \frac{N}{P} \ln\left(\frac{N}{M}\right) - \frac{3N}{4P} + \frac{NM}{P}\left(\frac{M-1}{M-2} - \frac{M^2}{2P} \frac{M-1}{M-2}\right) + 2, & \text{for } M < P \\
\frac{N^2}{MP} - \frac{N}{P} \ln\left(\frac{N}{M}\right) - \frac{N}{P}\left(\frac{2MP-M-2P}{2(P-1)} - \frac{M^2}{4P} - \frac{M^2/4 - M/2}{P-1} + 2\right), & \text{for } P = \frac{M}{P} \\
\frac{N^2}{MP} - \frac{N}{P} \ln\left(\frac{N}{M}\right) - \frac{N}{P}\left(\frac{7}{4} - M\right) - \frac{M^2}{2P} + \frac{M}{2P} + 2, & \text{for } M > \frac{P}{2}
\end{array} \right.
\]  

(6.3.2.11)

The \( S_{\text{total}} \) is now measured as follows:

\[
S_{\text{total}} = \left(\frac{N^2}{M^2} - \frac{1}{2} N \ln\left(\frac{N}{M}\right) + N(M - \frac{7}{4}) - \frac{1}{2} (M^2 - M)\right)/T_P,
\]

(6.3.2.12)

where \( T_P \) takes any value from equation (6.3.2.11) for the appropriate values of \( M \) and \( P \). The results of \( S_{\text{total}} \) obtained by applying equation (6.3.2.12) are listed in Table 6.7. These results are greater than those obtained experimentally and this is due to the same reasons as discussed for the case of the 2-way merge algorithm. In Table 6.7, we see that the merge speed-up obtained from the experimental results increases with the increase of the number of paths. However, the theoretical values of \( S_{\text{merge}} \) increases for all cases of \( M/2 \). This means the speed-up will remain steady for all three cases of \( M/2 \). On the other hand, \( S_{\text{total}} \) decreases when \( M \) increases but the theoretical values are mostly equal for different values of \( M \). The experimental values of the speed-up ratios
<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>No. of Paths</th>
<th>Time for sorting</th>
<th>Time for merging</th>
<th>Total time</th>
<th>Speed-up for sorting</th>
<th>Speed-up for merging</th>
<th>Total Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>88.050</td>
<td>1.970</td>
<td>90.020</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>47.630</td>
<td>1.300</td>
<td>48.930</td>
<td>1.85</td>
<td>1.52</td>
<td>1.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>23.330</td>
<td>1.280</td>
<td>24.610</td>
<td>3.77</td>
<td>1.54</td>
<td>3.66</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>43.090</td>
<td>4.130</td>
<td>47.220</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>23.280</td>
<td>2.620</td>
<td>25.900</td>
<td>1.85</td>
<td>1.58</td>
<td>1.82</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>16.910</td>
<td>2.060</td>
<td>18.970</td>
<td>2.55</td>
<td>2.00</td>
<td>2.49</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>11.640</td>
<td>1.510</td>
<td>13.150</td>
<td>3.70</td>
<td>2.74</td>
<td>3.59</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>20.970</td>
<td>8.360</td>
<td>29.330</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>11.260</td>
<td>4.850</td>
<td>16.110</td>
<td>1.86</td>
<td>1.72</td>
<td>1.82</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>8.680</td>
<td>3.680</td>
<td>12.360</td>
<td>2.41</td>
<td>2.27</td>
<td>2.37</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>5.890</td>
<td>2.800</td>
<td>8.690</td>
<td>3.56</td>
<td>2.98</td>
<td>3.37</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>10.250</td>
<td>17.090</td>
<td>27.340</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>5.670</td>
<td>9.500</td>
<td>15.170</td>
<td>1.81</td>
<td>1.80</td>
<td>1.80</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>4.220</td>
<td>7.710</td>
<td>11.930</td>
<td>2.43</td>
<td>2.55</td>
<td>2.29</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>2.950</td>
<td>5.330</td>
<td>8.280</td>
<td>3.47</td>
<td>3.21</td>
<td>3.30</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
<td>4.850</td>
<td>36.460</td>
<td>41.310</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>2.660</td>
<td>20.050</td>
<td>22.710</td>
<td>1.82</td>
<td>1.82</td>
<td>1.82</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>1.990</td>
<td>16.560</td>
<td>18.550</td>
<td>2.45</td>
<td>2.56</td>
<td>2.23</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>1.430</td>
<td>11.430</td>
<td>12.860</td>
<td>3.39</td>
<td>3.19</td>
<td>3.21</td>
</tr>
</tbody>
</table>

**TABLE 6.6**: The experimental results of Prog 6.3: The Bubble sort and odd-even merge algorithms
TABLE 6.7: The theoretical and experimental values of the speed-up for Prog 6.3 in Table 6.7 are presented graphically in Figure 6.5.

From the experimental results of Prog 6.3, we notice that the maximum number of paths that gives an optimum total speed-up for $N=1024$ and $P=4$ is when $M=32$, and for $N=512$ (results are not included) is when $M=16$. Therefore, we can find a value of $M$ that corresponds to the maximum decomposition of the problem for fixed values of $N$ and $P$. This number is given by,

$$M = \left( \frac{\log N}{2} \right) + 3 \times P.$$

Next, we want to find the maximum number of processors that one can activate to participate in the problem solution when a given number of paths
FIGURE 6.5
The Experimental Total and Merge Speed-up Values of Prog 6.3
is used. To predict this number, we list in Table 6.8 some theoretical values for the total speed-up where different values for M and P are given. Therefore, it is obvious from Table 6.8 that the best choice of the maximum number of processors to be used is given by,

\[ P \leq \frac{M}{2} \]

This is generally true since the nature of the merge algorithm is to halve the number of paths. Thus, when \( \frac{M}{2} = P \) all the paths will be carried out simultaneously. Therefore, better results may be obtained and a linear speed-up is obtained.

In conclusion, we compare the results obtained from the odd-even merge and the 2-way merge implementations. We observe that the merge speed-up in the odd-even merge is better than the 2-way merge speed-up. Also, the total speed-up \( S_{\text{total}} \) for the odd-even merge slightly decreases with the increasing M, whereas a decrease was significant in the 2-way merge method. The total times required by Prog 6.1 for the 2-way merge and that of Prog 6.3 for the odd-even merge are illustrated in Figures 6.6 and 6.7 respectively for different numbers of processors and different numbers of paths. These results are taken from Tables 6.1 and 6.6 respectively. In Figure 6.6, the curves for \( P=2,3,4 \) start to be close up at \( M=32 \) whilst in Figure 6.7 the difference is significant at this point.

The second difference between Figure 6.6 and 6.7 is that the run-times in the case of the 2-way merge method was slightly different than that of the odd-even method. This difference becomes significant as the number of paths increases. This is mainly due to the different times required for the merge procedure of the two methods. For illustration, we present in Figure 6.8 the difference in the timing results for the merge procedure alone for both methods, and Figure 6.9 illustrates the merge speed-up for both methods, where these results are taken from Tables 6.1 and 6.6.
FIGURE 6.6
The Total Timing Results of Prog 6.1 (2-Way Merge& Bubble Sort)
FIGURE 6.7

The Total Timing Results of Prog 6.3 (Odd-Even Merge & Bubble Sort)
<table>
<thead>
<tr>
<th>N</th>
<th>M</th>
<th>P</th>
<th>$S_{\text{total}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1024)</td>
<td>8</td>
<td>2</td>
<td>1.99</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>2.99</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>3.96</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td>4.88</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td>5.86</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
<td>6.84</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>7.81</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td></td>
<td>1.99</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>2.99</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>3.99</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td>4.99</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td>5.99</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
<td>6.99</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>7.97</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
<td>8.689</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td>9.65</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
<td>10.62</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td></td>
<td>1.99</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>2.99</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>3.99</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td>4.99</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td>5.99</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
<td>6.99</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>7.99</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
<td>8.689</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td>9.65</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
<td>10.62</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>15.69</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td></td>
<td>16.35</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td></td>
<td>17.31</td>
</tr>
</tbody>
</table>

TABLE 6.8: Theoretical values of the total speed-up (formula (6.3.2.12)) of Prog 6.3
 FIGURE 6.8

The Time Required For The Odd-Even and 2-Way Merge Methods
The Speed-up of The 2-Way and Odd-Even Merge Methods
From Figure 6.9, we observe that although the time for the odd-even merge was greater than that of the 2-way merge, the speed-up ratio for the former algorithm presents a significant difference over the speed-up ratio for the 2-way merge algorithm. Therefore, if we consider the complexity of both methods, the run-time results illustrate that the 2-way method implemented with the bubble sort for sorting and merging a particular data size is superior to the odd-even method implemented with the bubble sort. On the other hand, the difference in the merge speed-up for both methods in Figure 6.9 show that the odd-even merge method is better than the 2-way merge method for parallel implementation. This difference is mainly due to the usage of the processors. For instance, we showed previously how the number of processors is halved in each step of the 2-way merge whilst all the processors were mostly fully used in all the steps of the odd-even merge algorithm. This, in fact, is an important term in parallel implementation and how to exploit parallelism as fully as possible.

Now, we present the complexity measurements of Prog 6.4 that implements the odd-even merge algorithm with the neighbour sort algorithm. In order to measure the speed-up, we measure first the total complexity when the program runs on one processor. This complexity is the sum of the times of the neighbour sort (equation 6.2.2.1) and the odd-even merge given in equation (6.3.2.3). Thus,

\[
T_1 = N(\log \frac{N}{M}) - 1 + M + N(M-1) - \frac{4}{3}M^2 + \frac{1}{2}M
\]

\[
= N(\log \frac{N}{M}) + M - 2) - \frac{4}{3}M^2 + \frac{3}{2}M. \quad (6.3.2.13)
\]

In the measurements of the parallel implementation we sum up the results of equation (6.2.2) with that of equations (6.3.2.5) to (6.3.2.7). Therefore, we obtain $T_p$ as follows:
By dividing the results of equation (6.3.2.13) by that of (6.3.2.14), we obtain the total speed-up $S_{total}$ for Prog 6.4. Thus,

$$S_{total} \geq \frac{M(\log(M) + M + 3)}{T_p} \tag{6.3.2.15}$$

where $T_p$ takes the appropriate value from inequality (6.3.2.14). The $S_{total}$ is simplified as follows:

$$S_{total} \geq \begin{cases} 
\frac{P(M-2)(\log(M) + M + 2)}{(M-2)(\log(M) + M - 2)}, & \text{for } M < P \\
\frac{2P(P-1)(\log(M) + M - 2)}{2(P-1)(\log(M) + 2MP - M - 4P + 2)}, & \text{for } \frac{M}{2} = P \\
O(P), & \text{for } \frac{M}{2} > P.
\end{cases} \tag{6.3.2.15'}$$

The experimental timing results and speed-up of Prog 6.4 are listed in Table 6.9. Meanwhile, the theoretical values of $S_{total}$ are measured from inequality (6.3.2.15) and are listed in Table 6.10.

From Table 6.10, we notice that in the case of $P=4$, the speed-up increases as $M$ increases. However, the best choice of $M$ to give the linear speed-up is when $\frac{M}{2} > P$ which is also indicated from the inequality (6.3.2.15').

Meanwhile, for any value of $M$ the total speed-up is greater than that of Prog 6.2 of the 2-way merge and the neighbour sort algorithms which was of $O\left(\frac{P(\log N-1)}{\log(N) + 2P-2}\right)$. This however, corresponds to the odd-even merge that possesses a speed-up greater than that of the 2-way merge as discussed previously.
<table>
<thead>
<tr>
<th>No. of processors</th>
<th>No. of paths</th>
<th>Time for sorting (in sec.)</th>
<th>Time for merging (in sec.)</th>
<th>Total time (in sec.)</th>
<th>Speed-up for sorting</th>
<th>Speed-up for merging</th>
<th>Speed-up for both parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>5.350</td>
<td>1.970</td>
<td>7.320</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2.890</td>
<td>1.300</td>
<td>4.190</td>
<td>1.85</td>
<td>1.52</td>
<td>1.75</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1.460</td>
<td>1.280</td>
<td>2.740</td>
<td>3.65</td>
<td>1.54</td>
<td>2.67</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>4.600</td>
<td>4.130</td>
<td>8.730</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2.490</td>
<td>2.620</td>
<td>5.110</td>
<td>1.85</td>
<td>1.58</td>
<td>1.71</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>1.830</td>
<td>2.060</td>
<td>3.890</td>
<td>2.51</td>
<td>2.00</td>
<td>2.24</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>1.290</td>
<td>1.510</td>
<td>2.800</td>
<td>3.57</td>
<td>2.74</td>
<td>3.12</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>3.860</td>
<td>8.360</td>
<td>12.220</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>2.100</td>
<td>4.850</td>
<td>6.950</td>
<td>1.84</td>
<td>1.72</td>
<td>1.76</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>1.470</td>
<td>3.680</td>
<td>5.150</td>
<td>2.63</td>
<td>2.27</td>
<td>2.37</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>1.080</td>
<td>2.800</td>
<td>3.880</td>
<td>3.57</td>
<td>2.98</td>
<td>3.15</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>3.150</td>
<td>17.090</td>
<td>20.240</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>1.690</td>
<td>9.500</td>
<td>11.190</td>
<td>1.86</td>
<td>1.80</td>
<td>1.81</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>1.170</td>
<td>7.710</td>
<td>8.880</td>
<td>2.69</td>
<td>2.55</td>
<td>2.28</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>0.890</td>
<td>5.330</td>
<td>6.220</td>
<td>3.54</td>
<td>3.21</td>
<td>3.25</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
<td>2.550</td>
<td>36.460</td>
<td>39.010</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>1.380</td>
<td>20.050</td>
<td>21.430</td>
<td>1.85</td>
<td>1.82</td>
<td>1.82</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>0.960</td>
<td>16.560</td>
<td>17.520</td>
<td>2.66</td>
<td>2.56</td>
<td>2.23</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>0.750</td>
<td>11.430</td>
<td>12.180</td>
<td>3.40</td>
<td>3.19</td>
<td>3.20</td>
</tr>
</tbody>
</table>

**Table 6.9:** The experimental results of Prog 6.4: the Odd-even merge and the neighbour sort algorithms
<table>
<thead>
<tr>
<th>Data size (N)</th>
<th>Processors (P)</th>
<th>Paths (M)</th>
<th>$S_{total}$ Theoretical</th>
<th>$S_{total}$ Experimental</th>
<th>Efficiency ($E_p$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1024)</td>
<td>2</td>
<td>4</td>
<td>1.81</td>
<td>1.75</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>1.99</td>
<td>1.71</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>1.99</td>
<td>1.76</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>1.99</td>
<td>1.81</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>1.99</td>
<td>1.82</td>
<td>0.99</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td></td>
<td>2.99</td>
<td>2.24</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>2.99</td>
<td>2.37</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>2.99</td>
<td>2.28</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>2.99</td>
<td>2.23</td>
<td>0.99</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td>3.07</td>
<td>2.67</td>
<td>0.77</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>3.71</td>
<td>3.12</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>3.99</td>
<td>3.15</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>3.99</td>
<td>3.25</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>3.99</td>
<td>3.20</td>
<td>0.99</td>
</tr>
</tbody>
</table>

**TABLE 6.10:** Theoretical and experimental values of the total speed-up of Prog 6.4
From the results of the previous two sections, we conclude, by referring to Tables 6.1, 6.4, 6.6 and 6.9, that the implementations of the 2-way merge and the odd-even merge algorithms with the bubble sort algorithm present a speed-up which is much better than that of the implementations with the neighbour sort algorithm, although the neighbour sort possesses less time complexity than the bubble sort. On the other hand, the odd-even merge algorithm when implemented with the bubble sort and the neighbour sort algorithms illustrates better speed-up than that of the 2-way merge algorithm. Rather, the 2-way merge requires fewer steps, hence comparisons, than the odd-even merge algorithm does. In fact, the 2-way merge algorithm is said to be a faster parallel merge algorithm from the time complexity view point, but we prove that the odd-even merge algorithm achieves higher speed-up and more efficiency in a parallel environment. However, in the next section, we extend the 2-way merge algorithm so that a better speed-up can be achieved.

6.3.3 The Parallel New Merge Algorithm

In this section, a new parallel merge algorithm is introduced to merge \( M \) sorted subsets that are sorted by the bubble sort algorithm of section (6.2.1). By following Gavril [1975], the new algorithm called "new merge algorithm" is implemented on the NEPTUNE system as follows.

This algorithm is considered as an extension to the 2-way merge algorithm presented in Section (6.3.1). Suppose that \( M \) subsets \((s_1, s_2, \ldots, s_M)\) are to be merged then this merge procedure requires \((\log M + 1)\) steps to obtain the final sorted set. In each step, the number of paths is halved and consequently the number of co-operating processors are halved. However, the algorithm is performed so that in the first step, each of the \( M \) subsets is partitioned into \( M \) segments of size \((N/M)^k\) each.
Therefore, the first processor is assigned to the first segments of all the subsets and these segments are arranged in a local vector, say $x$, so that the first segment of $s_1$ is arranged in the first $(N/M^2)$ locations of $x$, the first segment of $s_2$ is arranged in second $(N/M^2)$ locations of $x$ and so on. Similarly, the second processor is assigned to the second segments of the $M$ subsets, the third processor to the third segments of the $M$ subsets and so on. Each processor, then, performs the merging of its own segments using the sequential standard 2-way merge algorithm.

In the second step, half of the number of paths of the previous step is generated. Therefore, the size of the segments is doubled and the processors proceed as in the previous step. The algorithm proceeds in a similar manner until the final step where only one path is generated and carried out by one processor to merge only two segments to obtain the final sorted list. The program of this method is not included in the Appendix A since it is similar to the 2-way merge code, instead we present in Figure 6.10 a flowchart to illustrate the algorithm and in Figure 6.11, a simple code of the algorithm is presented.

It was mentioned above that this algorithm requires $(\log(M)+1)$ steps to complete the merge of $M$ subsets. Since, unlike the 2-way merge algorithm, the new merge algorithm proceeds by carrying out the $M$ paths in the first step as described above then the number of paths is halved in each of the following steps. This means that in the second and the following steps, the new merge algorithm behaves in a similar manner to the 2-way merge which requires $\log(M)$ steps. Hence, the new merge algorithm requires $(\log M+1)$ steps.

For the actual performance of this algorithm on the NEPTUNE system, the new merge algorithm is implemented with the bubble sort and we call this implementation "Prog 6.5". In order to carry out the analysis of the
FIGURE 6.10: The flowchart of the New Merge Algorithm
LH = \log(M) + 1
DO 200 KK = 1, LH
NP = M / 2**(KK - 1)
$DOPAR 15 IP = 1, NP
C arrange the segments taken from the original vectors A or B into X vector.
JQ = KK - (KK/2) * 2
NELM = N / M
NL = NELM / M
IND = 2**(KK - 1) * NELM * (IP - 1) + 1
DO 26 II = 1, M
IST = NELM * (II - 1) + 2** (KK - 1) * NL * (IP - 1) + 1
IFN = IST + 2** (KK - 1) * NL - 1
IF (JQ .EQ. 0) GOTO 17
DO 10 IT = IST, IFN
   X (IND) = A (IT)
   IND = IND + 1
10   CONTINUE
   GOTO 26
17   DO 21 IT = IST, IFN
       X (IND) = B (IT)
       IND = IND + 1
21   CONTINUE
26   CONTINUE
C Now follows a code to merge M segments by the 2-way merge algorithm. If C JQ = 1 then the elements of X are copied into B vector and if JQ = 0, C the elements of X are copied into A for further processing.
C
15 $PAREND
200 CONTINUE

FIGURE 6.11: The code of the New Merge Algorithm
new merge algorithm, we consider first the single processor performance.

Since the code used to merge the segments is the sequential standard 2-way merge algorithm, therefore, on average \((2N/M-1)\) comparisons are required to merge two subsets of size \((N/M)\) each (see Section (6.3.1)).

The complete analysis is presented as follows:

In each step, \(\log(M)\) sequential stages are required to merge two subsets of equal sizes. Therefore, in the first step, \(M\) paths are generated where each path contains \(M\) segments of size \((N/M^2)\) each to be processed by \(\log(M)\) sequential stages. Thus, the first stage of the first step requires the following complexity.

In this stage, \(M/2\) paths are generated to merge the \(M\) segments of size \((N/M^2)\) each. This requires \(M/2(2N/M^2-1)\) comparisons. The second stage of the first step requires \(M/4(4N/M^2-1)\) comparisons. This is because, the size of the segments is doubled in each stage. This proceeds until the final stage within the first step which requires \(M(2\log(M))N -1)\) comparisons.

To sum up, let \(C_1\) denote the total comparisons required in the main first step of the algorithm in which \(M\) paths are generated. Thus,

\[
C_1 = M\left[\frac{M}{2} \frac{2N}{M} - 1\right] + \frac{M}{4} \left(\frac{4N}{M^2} - 1\right) + \ldots + \frac{M}{2\log(M)} \left(\frac{2\log(M)}{M^2} N - 1\right)
\]

\[
= N \log M - M^2 (1 - \frac{1}{M})
\]

\[
= N \log M - M(M-1) \text{ comparisons} \quad (6.3.3.1)
\]

In the second step of the algorithm, \(M/2\) paths are generated where each path requires \(\log M\) sequential stages also. However, the first stage merges \(M/2\) segments of size \((2N/M^2)\) each, the second stage merges \(M/4\) segments of size \((4N/M^2)\) each and so on. Therefore, the total complexity achieved in the second step is given by \(C_2\). Thus,

\[
C_2 = \frac{M}{2} \frac{4N}{M^2} - 1\right] + \frac{M}{4} \left(\frac{8N}{M^2} - 1\right) + \ldots + \frac{M}{2\log(M)} \left(\frac{2\log(M)}{M} N - 1\right)
\]

\[
= N\log M - \frac{M^2}{2} (1 - \frac{1}{M})
\]

\[
= N\log M - \frac{M^2}{M} (M-1) \text{ comparisons.} \quad (6.3.3.2)
\]
Finally, in the final step of the algorithm (i.e. the $(\log(M)+1)$th step), only one path, i.e. $\frac{M}{2^{\log M}}$ path, is generated that carries out $\log M$ stages where, in the first stage, $M/2$ segments of size $\left(\frac{2^{\log M}}{M} \cdot N\right)$ each are merged to yield $\frac{M}{2} \left(\frac{2^{\log M} \cdot N}{M} - 1\right)$ comparisons, and so on until the final stage that requires $\frac{M}{2^{\log M}} \left(\frac{2^{\log M} \cdot N}{M} - 1\right)$ comparisons. Therefore, the final step of the algorithm achieves the following complexity:

$$C_{\log(M)+1} = \frac{M}{2^{\log M}} \left[\frac{2^{\log M+1}}{M} \cdot N - 1\right] + \frac{M}{4} \left(\frac{2^{\log M+2}}{M^2} \cdot N - 1\right) + \ldots + \frac{M}{2^{\log M} \cdot N} \left(\frac{2^{\log M} \cdot N}{M} - 1\right)$$

$$= N \log M - \frac{M}{2^{\log M}} (M-1) \text{ comparisons} \quad (6.3.3.3)$$

The summation of the results in the equations (6.3.3.1), (6.3.3.2) and (6.3.3.3) yields the total complexity $C_{\text{dm}}$ of the new merge algorithm when it is performed in one processor. Thus,

$$C_{\text{dm}} = N \log M - M(N-1) + \sum_{i=1}^{M-1} N \log M - \frac{M}{2^{\log M}} (M-1)$$

$$= N \log M (\log M + 1) - (M-1)(2M-1) \text{ comparisons} \quad (6.3.3.4)$$

The total complexity $T_1$ of Program 6.5 of the bubble sort and the new merge algorithms when implemented on a single processor then becomes:

$$T_1 = \frac{N^2}{M} - \frac{M}{2} \left(\frac{N}{M} - 1\right) + N + N \log M (\log M+1) - (M-1)(2M-1). \quad (6.3.3.5)$$

On the other hand, the complexity of Program 6.5 when implemented on a parallel $P$ processors is obtained as follows:

In the merge algorithm, after $\log(M/P)$ steps the number of processors is halved as in the 2-way merge algorithm. It follows that the same discussion for the new merge algorithm as in the 2-way merge algorithm is applicable. Therefore, the total complexity $C_{\text{pm}}$ required in the new merge algorithm in parallel is:

$$C_{\text{pm}} = \frac{M}{P} \left[\frac{2^{\log M}}{M} \cdot N - 1\right] + \frac{M}{4} \left(\frac{2^{\log M}}{M^2} \cdot N - 1\right) + \ldots + \frac{M}{2^{\log M} \cdot N} \left(\frac{2^{\log M}}{M} \cdot N - 1\right)$$
The total complexity $T_P$ of Prog 6.5 becomes:

$$T_P \leq \frac{N^2}{MP} - \frac{N}{P} \ln \left( \frac{N}{M} \right) - \frac{N}{P} \ln M \cdot \log \left( \frac{M}{P} \right) + \frac{2N}{P} \log M \cdot \log \left( \frac{M}{P} \right) - \left( \frac{2M}{P} - \log \left( \frac{M}{P} \right) + \log M \right). \quad (6.3.3.6)$$

Now we can find the merge speed-up and the total speed-up ratios.

Thus,

$$S_{\text{merge}}(N) = \frac{C_{\text{merge}}}{C_{\text{PM}}}$$
It is further simplified to yield that \( S \text{merge} \) becomes of \( \mathcal{O}\left(\frac{\log M}{M}\right) \) which is clearly greater than the speed-up of the 2-way merge algorithm which is of the order \( \mathcal{O}\left(\frac{\log M}{M}\right) \). One can clearly notice that the \( S \text{merge} \) increases as \( M \) increases, whilst, the \( S \text{total} \) decreases as \( M \) increases. However, when \( M=P \) a total speed-up of almost \( \mathcal{O}(P) \) is obtained.

The timing experimental results obtained when Prog 6.5 was run on the NEPTUNE system where \( N=1024 \) elements were sorted are listed in Table 6.11. The theoretical and experimental values of the merge and the total speed-ups are presented in Table 6.12, and the experimental values for different numbers of paths are illustrated in Figure 6.12.

From Figure 6.12, we notice that the higher value of the total speed-up is achieved when \( M=P \) whereas the higher value of the merge speed-up is when \( M=32 \) in this particular problem. On the other hand, by comparing the graphs illustrated in Figures 6.3 and 6.12, we notice that the speed-ups in Figure 6.12 are greater than that of Figure 6.3. Therefore, although the new merge algorithm requires more time complexity than the 2-way merge algorithm the speed-up observed is much more.

So far, a description of three parallel merge algorithms have been presented and compared from the speed-up point of view. Now, a performance analysis of these methods is predicted together with their performance
measurements when run on the NEPTUNE system. With reference to section (5.4), we measure the static and dynamic losses of the parallel paths control and the shared data. However, in order to measure the static losses of the parallel paths control, we have to know the number of accesses made by the program to a path per the total number of operations performed in the path. Similarly, the static loss of the shared data is obtained by considering how many accesses to the shared data per the total number of the operations carried by one path. On the other hand, the dynamic losses of the parallel paths control are obtained directly from the results of the NEPTUNE system and we include here the results of 4 processors when run in parallel. These measurements are the cost of the waiting cycles the processor has spent because no parallel path is available to be carried out by that processor. These calculations are listed in Table 6.13.

The parallel path access rate in the odd-even merge is equal to one access per \( \frac{2N}{M} - 1 \) flops, since the complexity (the number of the floating point operations, flops) within a path is \( \frac{2N}{M} - 1 \) flops (comparisons in this case), and there is only 1 access to a path per \( \frac{2N}{M} - 1 \) flops. On the other hand in the 2-way merge algorithm where the number of comparisons is changed in each step of the algorithm, an average of the complexities is taken to represent the total complexity per path. Hence, since this algorithm requires \( M \) steps, therefore the average complexity per path is given as \( \frac{2N}{M} - 1 \) \( \log M \). This means that there is one access to a path per \( \frac{2N}{M} - 1 \) \( \log M \). Finally, in the new merge method, \( \frac{M}{2} \log M(\frac{2N}{M} - 1) \) operations are performed within each path and since in each of the \( (\log M + 1) \) steps of the algorithm the complexity is changed therefore, this method made one access per \( M \log M(\frac{2N}{M} - 1)(\log M + 1) \) flops. The losses obtained from the above access rates can be measured by using the results of Table 5.10 that includes the timing results of the parallel path access on the NEPTUNE system. These
<table>
<thead>
<tr>
<th>No.of processors (P)</th>
<th>No.of paths (M)</th>
<th>Time for sorting (sec)</th>
<th>Time for merging (sec)</th>
<th>Total time (sec)</th>
<th>Speed-up for sorting</th>
<th>Speed-up for merging</th>
<th>Speed-up for both parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>88.050</td>
<td>3.840</td>
<td>91.890</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>47.630</td>
<td>2.700</td>
<td>50.330</td>
<td>1.85</td>
<td>1.42</td>
<td>1.83</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>23.330</td>
<td>2.260</td>
<td>25.590</td>
<td>3.77</td>
<td>1.70</td>
<td>3.59</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>43.090</td>
<td>7.340</td>
<td>50.430</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>23.280</td>
<td>4.790</td>
<td>28.070</td>
<td>1.86</td>
<td>1.53</td>
<td>1.80</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>16.910</td>
<td>4.380</td>
<td>21.290</td>
<td>2.55</td>
<td>1.68</td>
<td>2.37</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>11.640</td>
<td>3.820</td>
<td>15.460</td>
<td>3.71</td>
<td>1.92</td>
<td>3.26</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>20.970</td>
<td>12.080</td>
<td>33.050</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>11.260</td>
<td>7.550</td>
<td>18.810</td>
<td>1.86</td>
<td>1.60</td>
<td>1.76</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>8.680</td>
<td>6.650</td>
<td>15.330</td>
<td>2.66</td>
<td>1.82</td>
<td>2.16</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>5.890</td>
<td>5.560</td>
<td>11.450</td>
<td>3.70</td>
<td>2.17</td>
<td>2.89</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>10.250</td>
<td>19.380</td>
<td>29.630</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>5.670</td>
<td>11.680</td>
<td>17.350</td>
<td>1.88</td>
<td>1.66</td>
<td>1.71</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>4.220</td>
<td>9.860</td>
<td>14.080</td>
<td>2.77</td>
<td>1.97</td>
<td>2.10</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>2.950</td>
<td>8.050</td>
<td>11.000</td>
<td>3.68</td>
<td>2.41</td>
<td>2.69</td>
</tr>
</tbody>
</table>

**TABLE 6.11:** The experimental timing results of Prog 6.5: the New Merge and the Bubble Sort algorithms
<table>
<thead>
<tr>
<th>Processors P</th>
<th>Paths M</th>
<th>( S_{\text{merge}} )</th>
<th>( S_{\text{total}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Theory</td>
<td>Experimental</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1.999</td>
<td>1.42</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.53</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.60</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.66</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>2.21</td>
<td>1.68</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2.33</td>
<td>1.82</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>2.40</td>
<td>1.97</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1.997</td>
<td>1.70</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2.28</td>
<td>1.92</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2.48</td>
<td>2.17</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>2.61</td>
<td>2.41</td>
</tr>
</tbody>
</table>

**TABLE 6.12:** The theoretical and experimental results of the \( S_{\text{merge}} \) and \( S_{\text{total}} \) of Prog 6.5
FIGURE 6.12
The Experimental Total and Merge Speed-up Values of Prog 6.5
(New Merge)
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Paths (M)</th>
<th>Shared data access rate</th>
<th>Shared data loss</th>
<th>Parallel path access rate</th>
<th>Parallel path loss</th>
<th>Parallel path loss static</th>
<th>Parallel path loss contention</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-way merge</td>
<td>4</td>
<td>2:1 flop</td>
<td>0.2%</td>
<td>1:((2N/M-1).logM) flops</td>
<td>0.11%</td>
<td>0.06%</td>
<td>2.6%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>0.14%</td>
<td>0.11%</td>
<td>2.26%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td>0.22%</td>
<td>0.19%</td>
<td>2%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td>0.36%</td>
<td>0.38%</td>
<td>1.79%</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td>0.61%</td>
<td>0.64%</td>
<td>1.58%</td>
</tr>
<tr>
<td>Odd-even merge</td>
<td>4</td>
<td>2:1 flop</td>
<td>0.2%</td>
<td>1:((2N/M-1) flops</td>
<td>0.13%</td>
<td>0.05%</td>
<td>1.35%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>0.45%</td>
<td>0.47%</td>
<td>1.05%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td>0.89%</td>
<td>0.86%</td>
<td>0.95%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td>1.81%</td>
<td>1.86%</td>
<td>0.5%</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td>3.68%</td>
<td>3.4%</td>
<td>0.38%</td>
</tr>
<tr>
<td>New merge</td>
<td>4</td>
<td>2:1 flop</td>
<td>0.2%</td>
<td>1:[Mlog(M*(2N/M-1)/2) M (logM+1)] flops</td>
<td>0.04%</td>
<td>0.06%</td>
<td>1.92%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>0.04%</td>
<td>0.07%</td>
<td>1.90%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td>0.05%</td>
<td>0.09%</td>
<td>1.81%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td>0.12%</td>
<td>0.14%</td>
<td>1.63%</td>
</tr>
</tbody>
</table>

**TABLE 6.13:** The performance measurements of the merge methods
losses are given as a percentage for different values of paths (M) when N was 1024. However, if we compare these losses with the static parallel path losses obtained from the NEPTUNE system when the methods are run on 4 processors, we notice that our prediction of the parallel path control comes within agreement of the experimental results. However, the contention in Table 6.13 cannot be predicted and can only be obtained from the results of the four processors performance.

It is clear that the contention for the parallel paths in the 2-way merge method is slightly greater than that of the new merge method and this is because, in the latter method, \( \log M + 1 \) steps are performed and always all the processors are in use in the first step of the algorithm. On the other hand, the odd-even method has the lowest contention losses amongst the three methods and obviously, this is related to the fact that this method uses all the processors in all the steps. Also, this loss decreases when M increases in which case there is always a path ready to be accessed.

The static shared data access loss is measured in a similar manner. Thus, the odd-even merge method has made \( 2 \left( \frac{2N}{M} - 1 \right) \) accesses to the shared data per \( \left( \frac{2N}{M} - 1 \right) \) flops, i.e. a rate of 2:1. The two-way merge method has made \( 2 \left( \log M \left( \frac{2N}{M} - 1 \right) \right) \) accesses per \( \left( \log M \left( \frac{2N}{M} - 1 \right) \right) \) flops which is also a rate of 2:1. Similarly, the new merge algorithm has a rate of 2:1 of the shared data accesses. Therefore, all the methods have a static shared data access loss equal to approximately 0.2%. This means that the three methods have the same amount of the static shared data losses but differ in their parallel path control losses according to how each method uses the processors to carry out the M paths.

### 6.3.4 Merging by the Binary Search Algorithm

The merging of two ordered files of size n and m was previously
proposed, Hwang and Lin [1972]. The implementation we shall discuss here concerns the merging of \( N \) subsets of equal sizes. Knuth [1973], however, studied extensively the search problem in general and here we present the code to merge \( A=(a_1, \ldots, a_m) \) with \( B=(b_1, \ldots, b_n) \) by the binary search algorithm:

(i) Set \( i=1 \)

(ii) Merge \( a_i \) into \( B \) by the binary search algorithm (BS below)

(iii) Pull out \( a_i \) and the elements of \( B< a_i \). (These are already in order and larger than the rest of the elements of \( A \cup B \)). Set \( n \) to the new value \( n' \). \( n' \) equals the value of the location of the first element of \( B \) which is greater than \( a_i \). Set \( i=i-1 \) and go back to step (i) until all the elements of \( A \) are inserted into \( B \).

The binary search algorithm (BS) is to insert an element \( k \) into \( A=(a_1, a_2, \ldots, a_n) \) by the following steps,

1. Set \( l=1 \) and \( u=n \)

2. If \( u<l \), the algorithm terminates unsuccessfully. Otherwise set \( i=\lfloor (l+u)/2 \rfloor \).

3. If \( k<a_i \), go to (4); if \( k>a_i \), go to (5), and if \( k=a_i \), the algorithm terminates successfully.

4. Set \( u=i-1 \) and go to (2)

5. Set \( l=i+1 \) and go to (2).

However, in our implementation there are \( M \) subsets of size \( (N/M) \) each to be merged to form one sorted set. The algorithm will proceed in \( \log M \) steps where the parallelism within each step is exactly as in the 2-way merge algorithm (Section 6.3.1).

In the worst case, each element of the first subset has to be compared \( \log N \) times if \( N \) is the size of the subset. Therefore, \( N \log N \) comparisons
are required to merge two subsets of size \( N \). This algorithm is implemented with the bubble sort that sorts the \( M \) subsets, and we denote such implementation as "Prog 6.6". The experimental results of Prog 6.6 are listed in Table 6.14 where the data size \( N \) is 1024 elements.

Now, we consider the expected complexity of the binary merge algorithm. In order to merge two subsets \( S_1 \) and \( S_2 \) by the binary merge algorithm, as mentioned earlier, after inserting, say, \( a_i \) of \( S_1 \) into \( S_2 \), the pointer will take the value of the position of the element where \( a_i \) has just been inserted. Therefore, the next element of \( S_1 \) to be inserted is to be compared with the elements of \( S_2 \) starting from the pointer obtained from the previous insertion. Since the input data is chosen at random in the interval \([0,1]\), the elements are uniformly distributed. Therefore, after inserting an element into \( S_2 \) the probability of the pointer to be located at any position is \((1/s)\), where \( s \) is the number of elements in the subset. The probability that the segment between the pointer and the final element in \( S_2 \) consists of \( i \) elements is obtained from the Binomial distribution,

\[
P(i) = \binom{s}{i} \left( \frac{1}{s} \right)^i \left( 1 - \frac{1}{s} \right)^{s-i}.
\]  

Therefore, we merge the remaining elements with a subset of \( i \) elements.

The expected number of comparisons in this case is given by:

\[
\sum_{i=1}^{s} T(i) \cdot P(i) ,
\]

and the total comparisons of the \( s \) elements is given by,

\[
\sum_{i=1}^{s} s \cdot T(i) \cdot P(i) .
\]  

(6.3.4.2)

Since, in order to merge \( M \) subsets, \( \log M \) steps are required such that the number of paths is halved in each step. Therefore, if we consider the total complexity of each step as a recurrence equation, then,

\[
T(s) = s \sum_{i=1}^{s} \binom{s}{i} \left( \frac{1}{s} \right)^i \left( 1 - \frac{1}{s} \right)^{s-i} \cdot T(i) ,
\]  

(6.3.4.3)

is the complexity of a step which is performed on subsets of size \( s \) each.
<table>
<thead>
<tr>
<th>No. of processors</th>
<th>No. of paths</th>
<th>Time for sorting</th>
<th>Time for merging</th>
<th>Total time</th>
<th>Speed-up for sorting</th>
<th>Speed-up for merging</th>
<th>Speed-up for both parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>88.050</td>
<td>9.950</td>
<td>98.00</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>47.630</td>
<td>7.750</td>
<td>55.380</td>
<td>1.85</td>
<td>1.28</td>
<td>1.77</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>23.330</td>
<td>7.680</td>
<td>31.010</td>
<td>3.77</td>
<td>1.30</td>
<td>3.16</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>43.090</td>
<td>14.010</td>
<td>57.100</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>23.280</td>
<td>9.910</td>
<td>33.190</td>
<td>1.86</td>
<td>1.41</td>
<td>1.72</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>16.910</td>
<td>9.790</td>
<td>26.700</td>
<td>2.55</td>
<td>1.43</td>
<td>2.14</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>11.640</td>
<td>8.830</td>
<td>20.470</td>
<td>3.71</td>
<td>1.59</td>
<td>2.79</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>20.970</td>
<td>17.440</td>
<td>38.410</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>11.260</td>
<td>11.740</td>
<td>23.000</td>
<td>1.85</td>
<td>1.49</td>
<td>1.67</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>8.680</td>
<td>11.170</td>
<td>19.850</td>
<td>2.64</td>
<td>1.56</td>
<td>1.94</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>5.890</td>
<td>9.790</td>
<td>15.680</td>
<td>3.63</td>
<td>1.78</td>
<td>2.45</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>10.250</td>
<td>20.370</td>
<td>30.620</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>5.670</td>
<td>13.290</td>
<td>18.960</td>
<td>1.88</td>
<td>1.53</td>
<td>1.62</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>4.220</td>
<td>12.280</td>
<td>16.500</td>
<td>2.74</td>
<td>1.66</td>
<td>1.86</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>2.950</td>
<td>10.530</td>
<td>13.480</td>
<td>3.60</td>
<td>1.93</td>
<td>2.27</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
<td>4.850</td>
<td>22.720</td>
<td>27.570</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>2.660</td>
<td>14.520</td>
<td>17.180</td>
<td>1.87</td>
<td>1.56</td>
<td>1.61</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>1.990</td>
<td>13.080</td>
<td>15.070</td>
<td>2.72</td>
<td>1.74</td>
<td>1.83</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>1.430</td>
<td>11.130</td>
<td>12.560</td>
<td>3.58</td>
<td>2.04</td>
<td>2.20</td>
</tr>
</tbody>
</table>

**TABLE 6.14:** The experimental results of Prog 6.6: The Bubble Sort and the Binary Search Merge algorithms
The number of paths carried out in each step is ignored for the moment.

Equation (6.3.4.3) is equivalent to

\[ T(s) = \sum_{i=1}^{s} \left( \frac{s}{i} \right) (s-1)^{s-i} \cdot \frac{1}{i} (s-1)^{s-1} \cdot T(i) \]  

(6.3.4.4)

Let \( T(1) = c_0 \) then equation (6.3.4.4) becomes,

\[ T(s) = c_0 \cdot s \left( \frac{1}{s} \right) (s-1)^{s-1} + s! \left( \frac{1}{s} \right) (s-1)^{s-1} \cdot \sum_{i=2}^{s} \frac{T(i)}{i! (s-i)! (s-1)^i} \]

Now, for \( 1 < i \leq s \) let,

\[ W_i = \frac{T(i)}{i! (s-1)! (s-1)^i} \]

Then,

\[ W_{i+1} = \frac{T(i+1)}{W_i} \cdot \frac{(s-i)}{i+1 (s-1)} \]

(6.3.4.5)

Since \( T(i) \cdot \log(i) \) (the binary search complexity of \( i \) elements). Therefore we have,

\[ \frac{T(i+1)}{T(i)} \leq \frac{\log(i+1)}{\log(i)} \]

Thus, if we substitute this result into equation (6.3.4.6) we obtain:

\[ W_{i+1} < W_i \cdot \frac{(s-i)}{(i+1)(s-1)} \cdot \frac{\log(i+1)}{\log(i)} < W_i \cdot \frac{1}{i+1} \cdot \frac{\log(i+1)}{\log(i)} \]

By iterating on \( W_{i+1} \), we then obtain,

\[ W_i < W_2 \cdot \frac{2}{i!} \log i, \text{ for } i > 2 \]

(6.3.4.7)

Now, substitute equation (6.3.4.7) into equation (6.3.4.6), we obtain:

\[ T(s) < c_0 \cdot s \left( \frac{s-1}{s} \right) (s-1)^{s-1} + s! \left( \frac{s-1}{s} \right) (s-1)^{s-1} \cdot \sum_{i=2}^{s} \frac{1}{i!} \cdot \frac{\log(i+1)}{\log(i)} \]

(6.3.4.8)

The value of \( W_2 \) can be obtained easily. Thus,

\[ W_2 = \frac{T(2)}{(s-2) ! (s-1)} \leq \frac{1}{2(s-1)! (s-1)} \]

since \( T(2) = \log(2) = 1 \).

Equation (6.3.4.8) then becomes,

\[ T(s) < c_0 \cdot s \left( \frac{s}{s-1} \right) \left( \frac{s-1}{s} \right)^{s-1} + s \cdot s! \left( \frac{s-1}{s} \right) (s-1)^{s-1} \cdot \sum_{i=2}^{s} \frac{\log i}{i!} \]

(6.3.4.9)

Since \( \left( \frac{s-1}{s} \right)^{s} \leq \left( 1 - \frac{1}{s} \right)^{s} < \frac{1}{e} \) (because, \( \lim_{n \to \infty} (1- \frac{1}{n})^n = \frac{1}{e} \)), and since
The series \( \sum_{i=2}^{\infty} \frac{\log i}{i!} \) is less than \( \sum_{i=2}^{\infty} \frac{1}{i!} = e \), therefore equation (6.3.4.9) can be
written as:

\[
T(s) < C_0 \frac{s}{s-1} \cdot \frac{1}{e} + s\left(\frac{s}{s-1}\right),
\]

or

\[
T(s) < s\left(\frac{s}{s-1}\right) \cdot \frac{C_0}{e} + 1.
\]  

Equation (6.3.4.10) represents the general solution of the expected average complexity of any step in the binary merge algorithm. Since our implementation considers \( M \) subsets of size \( \frac{N}{M} \) each to be merged in \( \log M \) steps. Then, for the one processor performance, the following complexities are obtained where the results of equation (6.3.4.10) are applied. Thus, the first step’s complexity \( T_{11} \) where \( M/2 \) paths are generated is given by:

\[
T_{11} < \frac{M}{2} \cdot \frac{N}{M} \cdot \frac{C_0}{e} + 1.
\]  

In the second step, each two subsets of size \( \frac{2N}{M} \) are merged where \( M/4 \) paths are required. In this case, the complexity \( T_{12} \) is given as:

\[
T_{12} < \frac{M}{4} \cdot \frac{2N}{M} \cdot \frac{C_0}{e} + 1.
\]

We proceed until the final step where \( \frac{M}{2 \log M} \) paths are generated to merge subsets of size \( \frac{2}{(\log M - 1)} \cdot \frac{N}{2} \) each. Hence, the complexity of this step is:

\[
T_{l \log M} < \frac{M}{2 \log M} \cdot \frac{N}{M} \cdot \frac{C_0}{e} + 1.
\]

From the summation of the complexities of all the steps, we obtain the total complexity of the serial implementation \( T_{1M} \) of the binary merge algorithm. Thus,

\[
T_{1M} = T_{11} + T_{12} + \ldots + T_{l \log M} < \frac{N}{2} \cdot \left(\frac{C_0}{e} + 1\right) \cdot \left[ \frac{N}{M-1} + \frac{2N}{2N/M-1} + \ldots + \frac{2^{\log M - 1}}{2^{\log M - 1}} \cdot \frac{N}{M-1} \right]
\]

\[
< \frac{3}{2} \cdot \frac{N}{2} \cdot \log M \cdot \log M.
\]

since each term of the form \( \frac{s}{s-1} \) is approximately 1 provided that \( s \gg 1 \) and \( C_0 = 1 \).

In the parallel implementation, we proceed in a similar manner to
the 2-way merge algorithm where the number of processors $P$ is halved after
the $\log(P)$th step. Therefore, the expected average complexity $T_{PM}$ in
parallel is:

$$T_{PM} < \frac{M}{2P} \left( \frac{N}{M} \right)^{\log(M/P) - 1} M \left( \frac{N}{M} \right)^{\log(M/P) - 1} M + \frac{M}{2P} \left( \frac{N}{M} \right)^{\log(M/P) + 1} M \left( \frac{N}{M} \right)^{\log(M/P) + 1} M + \ldots $$

$$+ \left( \frac{M}{2P} \right)^{\log(M/P) + 1} M \left( \frac{N}{M} \right)^{\log(M/P) + 1} M + \ldots $$

The merge speed-up is then obtained as,

$$S_{\text{merge}} = \frac{T_{1M}}{T_{PM}} = \frac{3N}{2P} \log M $$

$$< \frac{3N}{2P} \log M + \frac{3N}{2P} (P-1) + \log M $$

(6.3.4.15)

It is easy to show that $S_{\text{merge}}$ is of $O\left( \frac{P \log M}{\log \frac{M}{P} + 2P-2} \right)$ which is approximately
what the experimental results have achieved, and also equal to the speed-up
of the 2-way merge algorithm.

As Prog 6.6 implements the bubble sort with the binary merge algorithm,
therefore, by considering the complexity of the bubble sort given in
Section (6.2.1), the total speed-up $S_{\text{total}}$ can be obtained. Thus,

$$S_{\text{total}} = \frac{T_{1}}{T_{P}}$$
One can check that when $M$ increases the $T_P$ of inequality (6.3.4.17) decreases, hence the second term of this inequality increases and $S_{\text{total}}$ becomes less than $P$. Therefore, we conclude that $S_{\text{total}}$ decreases as $M$ increases. This is clearly seen in Table 6.15 where the theoretical results of $S_{\text{merge}}$ and $S_{\text{total}}$ obtained from the inequalities (6.3.4.16) and (6.3.4.17) respectively are listed.

We notice from Table 6.15 that when $M=P=4$ the merge speed-up equals to that when $P=2$ and this is because when $M=4$ means only two paths are generated in the first step, hence only two processors co-operate in the evaluation. In general, $S_{\text{merge}}$ of this algorithm is less than that of the 2-way merge and also the running time of the binary merge algorithm is greater than that of the 2-way merge. Thus, the total speed-ups become less.

6.3.5 Merging by a Parallel Jump Searching Algorithm

In this section, merging by the jump searching or (block searching) algorithm is presented. The jump searching algorithm is described extensively by Shneiderman [1978] who followed the work done by Martin [1977]. This algorithm is applied in some situations when the binary search algorithm is unsuitable. One of the cases where the jump searching algorithm is applied is when the keys of the records in a file have been compressed producing variable length keys and in this situation the binary search algorithm cannot be applied. In some other situations where the

\[
S_{\text{total}} < \frac{3 N^2}{4 M} - \frac{1}{2} N \ln \left( \frac{N}{M} \right) - \frac{3 N}{4} + \frac{3N}{4} \log M \\
\frac{3 N^2}{4 M} - \frac{1}{2} N \ln \left( \frac{N}{M} \right) - \frac{3 N}{4} + \frac{3N}{4p} \log \left( \frac{M}{P} \right) + \frac{3N}{2p} (P-1) + \log M
\]

\[
< P \left( 1 - \frac{N(P-1)}{P} + \log M \right).
\]

(6.3.4.17)
<table>
<thead>
<tr>
<th>N</th>
<th>P</th>
<th>M</th>
<th>S\textsubscript{merge}</th>
<th>S\textsubscript{total}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Theory</td>
<td>Experimental</td>
</tr>
<tr>
<td>(1024)</td>
<td>2</td>
<td>4</td>
<td>1.33</td>
<td>1.28</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>1.49</td>
<td>1.41</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>1.60</td>
<td>1.49</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>1.66</td>
<td>1.53</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>1.71</td>
<td>1.56</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8</td>
<td>1.66</td>
<td>1.43</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>1.86</td>
<td>1.56</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>2.01</td>
<td>1.66</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>2.13</td>
<td>1.74</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>1.33</td>
<td>1.30</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>1.71</td>
<td>1.59</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>1.99</td>
<td>1.78</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>2.21</td>
<td>1.93</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>2.39</td>
<td>2.04</td>
</tr>
</tbody>
</table>

**TABLE 6.15:** The theoretical values of $S_{\text{total}}$ and $S_{\text{merge}}$ of Prog 6.6
The jump searching algorithm is preferred in sorted files which are linked to maintain a sequential ordering. This may be the case for the files of variable length records.

Various types of the jump searching algorithm in which the jump size is different were proposed by Shneiderman, and these are as follows:

(1) The simple jump searching algorithm:

This method requires jumps of the size of the square root of the number of records in the file. However, in order to demonstrate that the optimum jump size is the square root of the number of keys, we need to determine the average expected cost of the search $C_1(N)$ over $N$ elements:

$$C_1(N) = \frac{1}{N/n} + \frac{1}{(n-1)} ,$$

(6.3.5.1)

where $n$ is the jump size. $C_1(N)$ can be read as, on average we have $\frac{1}{N/n}$ jump searches and the sequential search is performed within the block of size $(n-1)$ that is found to contain the key under consideration.

If we assume that $C_1(N)$ is a continuous function, then we can take the derivative with respect to $n$. This yields:

$$C'_1(N) = -\frac{1}{N/n^2} + \frac{1}{(n-1)} .$$

(6.3.5.2)

By setting equation (6.3.5.2) to zero, we obtain,

$$n = \sqrt{N} .$$

Therefore, the total number of comparisons becomes:

$$C_1(N) = \frac{1}{N/\sqrt{N}} + \frac{1}{(\sqrt{N}-1)} = \sqrt{N} - \frac{1}{2} \approx \sqrt{N} \text{ comparisons.}$$

(2) Two level simple jump searching

In this method the jump searching algorithm can be applied at two levels. In other words, jumps may be applied within the block that contains the target key where the square root jump size is assumed and reapplied to the $(N-1)$ keys in that block. This method is considered to be twice as fast as the simple jump searching method as the following calculations prove:
In this method, the first jumps are of size $\sqrt{N}$ and the second jumps are of size $N^{1/3}$. Since this second jump is done within the block of the first jump, therefore, we have:

$$C_2(N) = \frac{1}{4} \frac{N}{\sqrt{N}} + \frac{1}{4} \frac{\sqrt{N}}{N} + \frac{1}{4} (N^{1/3} - 1)$$

$$= \frac{1}{4} \sqrt{N} + \frac{1}{4} N^{1/3} \text{ comparisons,}$$

which is about half of $C_1(N)$.

(3) Two level fixed jump searching

If we allow two levels of jump searching then it is no longer optimal to have the first jump as small as $\sqrt{N}$. If we assume that the first level jumps are of size $n_1$ and the second level jumps of size $n_2$ then the approximate search cost for a two level fixed jump search algorithm is given by:

$$C_3(N) = \frac{4N}{n_1} + \frac{4n_1}{n_2} + \frac{4}{4} (n_2 - 1) \quad (6.3.5.3)$$

Also, we assume this function is continuous. Hence, by taking the partial derivatives with respect to $n_1$ and $n_2$ we obtain:

$$\frac{\partial C_3}{\partial n_1} = -\frac{4N}{n_1^2} + \frac{4}{2n_2},$$

$$\frac{\partial C_3}{\partial n_2} = -\frac{4n_1}{n_2^2} + \frac{4}{4} \quad (6.3.5.4)$$

By setting the two equations in (6.3.5.4) to zero and solving, we obtain:

$$n_1 = N^{2/3}, \quad \text{and} \quad n_2 = N^{1/3}.$$

If these values are placed in equation (6.3.5.3) we obtain:

$$C_3(N) = \frac{3}{2} N^{1/3} \text{ comparisons.} \quad (6.3.5.5)$$

Therefore, this method is faster than the other two methods.

In our implementation on the NEPTUNE system we choose methods (2) and (3) mentioned earlier to merge M sorted subsets that are to be sorted by the bubble sort algorithm.
The Two-Level Simple Jump Merging Method

This method is applied to merge M sorted subsets of size \((N/M)\) each where the elements of one subset are considered as target elements in the second subset. Since the merge can be performed on only two subsets at a time, therefore, \(\log M\) steps are required to perform the merge to obtain the final sorted list. However, to merge two subsets \(S_1\) and \(S_2\), the first element \(a_1\) of \(S_1\) is compared with the element \(b\) of \(S_2\) which is in the position \(\sqrt{s}\), where \(s\) is the size of the subsets. If \(b\) is greater than \(a_1\) then another level of jumps is performed. This means that \(a_1\) is compared with the element at the position of \((s')\). After these two levels of jumps a sequential searching is performed until the required position of \(a_1\) is found. The segment before that which \(a_1\) is inserted will not be scanned any more and the scan of the second element \(a_2\) will be started after that segment. This means that the size \(s\) will be modified after each insertion of the elements. Therefore, those elements at the end of the subset will require a very small number of comparisons or may be not one in the case when the segment is the last segment in \(S_2\). This method is implemented in Program A.9 shown in Appendix A.

Let us call this method "Prog 6.7" whose experimental results are listed in Table 6.16. For the analysis of the complexity of this method we proceed as follows:

Since the input data is generated randomly in the interval \([0,1]\) according to an uniform distribution, therefore, the probability of inserting an element of \(S_1\) into any position of \(S_2\) is \(\left(1 - \frac{1}{N/M}\right)\). Also, the probability of the distance between this last insertion and the final element of \(S_2\) which is \(i\) is obtained from the binomial distribution

\[
P(i) = \binom{N/M - 1}{i} \left(1 - \frac{M}{N}\right)^{(N/M-1)}.
\]

Therefore, to merge an element with a subset of size \(i\) using the jump
searching algorithm requires the following average number of comparisons:

$$\frac{N}{M} \sum_{i=1}^{\frac{N}{M}} (\frac{1}{i})^\frac{1}{\alpha} P(i) .$$  \hspace{1cm} (6.3.5.6)

Equation (6.3.5.6) can be written as a recurrent equation and we can proceed as in the binary merge algorithm. Then we have:

$$T(s) = \sum_{i=1}^{s} \left( \frac{s-1}{s} \right)^{(s-1)} \frac{1}{i} \sum_{i=2}^{s} \frac{T(i)}{i! \cdot (s-1)! \cdot (s-1)!} .$$

Let $T(1) = c_0$, then,

$$T(s) = c_0 \sum_{i=1}^{s} \left( \frac{s-1}{s} \right)^{(s-1)} \frac{1}{i} \sum_{i=2}^{s} \frac{T(i)}{i! \cdot (s-1)! \cdot (s-1)!} .$$

Let

$$W_1 = \frac{T(i)}{i! \cdot (s-1)! \cdot (s-1)!} .$$

Then,

$$W_{i+1} = W_i \frac{T(i+1)}{T(i)} \cdot \frac{(s-1)}{(i+1)(s-1)} < W_i \frac{1}{i+1} \frac{T(i+1)}{T(i)} .$$

Since $T(i) \leq (\frac{1}{i})^\frac{1}{\alpha} + \frac{1}{i}^\frac{1}{\alpha}$, then,

$$W_{i+1} < W_i \frac{1}{i+1} \frac{(\frac{1}{i})^\frac{1}{\alpha} + (i+1)^\frac{1}{\alpha}}{\frac{1}{i}^\frac{1}{\alpha} + i} .$$

Thus,

$$W_1 < W_2 \frac{2(\frac{1}{i})^\frac{1}{\alpha} + (i+1)^\frac{1}{\alpha}}{i! \cdot (\frac{1}{i})^\frac{1}{\alpha} + (i+1)^\frac{1}{\alpha}} .$$

Therefore,

$$T(s) < s \cdot \left( \frac{s-1}{s} \right)^{(s-1)} + s \cdot s! \cdot \left( \frac{s-1}{s} \right)^{(s-1)} \cdot \sum_{i=2}^{s} \frac{W_2 \cdot \frac{1}{i}^\frac{1}{\alpha} + (i+1)^\frac{1}{\alpha}}{i!} .$$

Since,

$$W_2 = \frac{T(2)}{2! \cdot (s-2)! \cdot (s-1)^2} = \frac{1}{2} \cdot \frac{(\frac{1}{2})^\frac{1}{\alpha} + (2)^\frac{1}{\alpha}}{2 \cdot (s-1)! \cdot (s-1)!} .$$

Therefore,

$$T(s) < s \cdot \left( \frac{s-1}{s} \right)^{(s-1)} + s \cdot s! \cdot \left( \frac{s-1}{s} \right)^{(s-1)} \cdot \frac{2}{(\frac{1}{2})^\frac{1}{\alpha} + (2)^\frac{1}{\alpha}} \cdot \frac{(\frac{1}{2})^\frac{1}{\alpha} + (2)^\frac{1}{\alpha}}{2 \cdot (s-1)! \cdot (s-1)!} \cdot \sum_{i=2}^{s} \frac{1}{i!} .$$

Since,

$$\sum_{i=2}^{\infty} \frac{1}{i!} < \sum_{i=2}^{\infty} \frac{1}{i!} < \sum_{i=1}^{\infty} \frac{1}{i!} < e.$$  \hspace{1cm} (6.3.5.7)

Therefore,

$$T(s) < \frac{s^2}{(s-1)} \cdot \frac{1}{e} + \frac{s^2}{(s-1)} \cdot \frac{1}{e} + \frac{s^2}{(s-1)} \cdot \frac{1}{e} + 1,$$

$$< \frac{3}{2} \cdot s \cdot \frac{s+1}{s} .$$  \hspace{1cm} (6.3.5.7)
<table>
<thead>
<tr>
<th>No. of processors</th>
<th>No. of paths</th>
<th>Time for sorting</th>
<th>Time for merging</th>
<th>Total time</th>
<th>Speed-up for sorting</th>
<th>Speed-up for merging</th>
<th>Total Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>88.050</td>
<td>5.070</td>
<td>93.120</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>43.030</td>
<td>7.590</td>
<td>50.620</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>23.280</td>
<td>5.040</td>
<td>28.320</td>
<td>1.85</td>
<td>1.51</td>
<td>1.79</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>16.910</td>
<td>5.030</td>
<td>21.940</td>
<td>2.55</td>
<td>1.51</td>
<td>2.31</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>11.640</td>
<td>4.390</td>
<td>16.030</td>
<td>3.70</td>
<td>1.73</td>
<td>3.16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>20.970</td>
<td>10.330</td>
<td>31.300</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>11.260</td>
<td>6.400</td>
<td>17.660</td>
<td>1.86</td>
<td>1.61</td>
<td>1.77</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>8.680</td>
<td>6.060</td>
<td>14.740</td>
<td>2.41</td>
<td>1.70</td>
<td>2.12</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5.890</td>
<td>5.110</td>
<td>11.00</td>
<td>3.56</td>
<td>2.02</td>
<td>2.85</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>10.250</td>
<td>13.00</td>
<td>23.250</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>5.670</td>
<td>7.720</td>
<td>13.390</td>
<td>1.81</td>
<td>1.68</td>
<td>1.74</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>4.220</td>
<td>7.020</td>
<td>11.240</td>
<td>2.43</td>
<td>1.85</td>
<td>2.07</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2.950</td>
<td>5.720</td>
<td>8.670</td>
<td>3.47</td>
<td>2.27</td>
<td>2.68</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>64</td>
<td>4.850</td>
<td>15.760</td>
<td>20.610</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2.660</td>
<td>9.130</td>
<td>11.790</td>
<td>1.82</td>
<td>1.73</td>
<td>1.74</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1.990</td>
<td>8.020</td>
<td>10.010</td>
<td>2.45</td>
<td>1.97</td>
<td>2.06</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1.430</td>
<td>6.480</td>
<td>7.910</td>
<td>3.39</td>
<td>2.43</td>
<td>2.61</td>
</tr>
</tbody>
</table>

**TABLE 6.16:** The experimental results of Prog 6.7: The Two-level Simple Jump Merge and the Bubble Sort algorithm
Since \( \log M \) steps are required to complete the merge where the same concepts of the 2-way merge algorithm are applied therefore, for uni-
processor's implementation, the complexity of each step of the algorithm is obtained by applying the inequality (6.3.5.7) and considering the number of paths and the size of the subsets in each step. Thus, the complexity \( T_{11} \) of the first step is:

\[
T_{11} < \frac{3}{2} \frac{N (N/M)}{2} \frac{N/M}{N/M-1} .
\]

the complexity \( T_{12} \) of the second step is:

\[
T_{12} < \frac{3}{2} \frac{M (2N/M)}{4} \frac{2N/M}{(2N/M-1)} ,
\]

and so on until the final step that requires the complexity \( T_{1 \log M} \):

\[
T_{1 \log M} < \frac{3}{2} \frac{M}{2} \frac{2 \log M-1}{M} \frac{N}{2} \frac{2 \log M-1}{(N/M)} \frac{N/M}{N/M-1} .
\]

By summing up the complexities of all the steps, we obtain the total complexity \( T_{1M} \) of the sequential implementation. Thus,

\[
T_{1M} = T_{11} + T_{12} + \cdots + T_{1 \log M} < \frac{3}{2} \frac{N}{2} \frac{N/M}{(N/M-1)} + \frac{2N/M}{(2N/M-1)} + \cdots + \frac{2 \log M-1}{(2 \log M-1)} \frac{N/M}{N/M-1} .
\]

Since \( \frac{x}{x-1} \) is approximately 1 therefore,

\[
T_{1M} < \frac{3N}{4} \log M . \quad (6.3.5.8)
\]

On the other hand, in the parallel implementation, the complexity is derived as in the binary merge algorithm and is given by:

\[
T_{PM} < \frac{3N}{4P} \log \left( \frac{M}{P} \right) + \frac{3N}{2P} (P-1) + \log M . \quad (6.3.5.9)
\]

The merge speed-up then becomes:

\[
S_{merge} < \frac{\frac{3N}{4} \log M}{\frac{3N}{4P} \log \left( \frac{M}{P} \right) + \frac{3N}{2P} (P-1) + \log M} .
\]

\[
= \frac{P \log M}{(\log \left( \frac{M}{P} \right) + 2P-2)} \left[ 1 - \frac{\log M}{T_{PM}} \right] . \quad (6.3.5.10)
\]

The \( S_{merge} \) for the jump search algorithm is also of \( O\left( \frac{\log M}{\log \left( \frac{M}{P} \right) + 2P-2} \right) \).
The theoretical values of $S_{\text{merge}}$ obtained from inequality (6.3.5.10) are listed in Table 6.17 where only the cases $P=2,4$ are considered.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$P$</th>
<th>$M$</th>
<th>$S_{\text{merge}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Theory</td>
</tr>
<tr>
<td>(1024)</td>
<td>4</td>
<td>4</td>
<td>1.33</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.71</td>
<td>1.73</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.99</td>
<td>2.02</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>2.22</td>
<td>2.27</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>2.39</td>
<td>2.43</td>
</tr>
</tbody>
</table>

| 2   | 4   | 1.33| 1.33               |
|     | 8   | 1.50| 1.51               |
|     | 16  | 1.60| 1.61               |
|     | 32  | 1.66| 1.68               |
|     | 64  | 1.71| 1.73               |

**TABLE 6.17:** The theoretical values of the $S_{\text{merge}}$ of Prog 6.7

The results of Table 6.17 show a good agreement of the theoretical results of the $S_{\text{merge}}$ with those given by the experiments.

When the bubble sort is implemented with this merge algorithm the total speed-up becomes:

$$S_{\text{total}} < \frac{\frac{3N^2}{M} - \frac{3N \ln(N)}{M} - \frac{4N}{4P} \ln M}{\left(\frac{3N^2}{4PM} - \frac{N \ln(N)}{P} - \frac{3N}{4P} + \frac{3N(P-1)+\log M}{2P}\right)}$$

$$< \frac{\frac{N}{2}(2P-1) - \frac{N}{2} \log M - P \log M}{T_P}, \quad (6,3.5.11)$$

which is of $O(P)$. As $M$ increases $S_{\text{total}}$ decreases as indicated by the theoretical values of $S_{\text{total}}$ which are listed in Table 6.18. It is clear from Table 6.18 that the efficiency which is $\frac{S_{\text{total}}}{P}$ decreases as $P$ increases.
and this is due to the data communication and path allocation overheads.

Also, from Table 6.18, one can derive a formula for the maximum number of processors for which the speed-up is linear, i.e. of $O(P)$. If a particular number of paths is used for a given data size, the number of processors is then given by:

$$P \leq \left\lfloor \frac{\log (N/M)}{2} \right\rfloor.$$

However, by referring to Table 6.16 of the experimental results of Prog 6.7, we can derive the empirical formula for the best choice of the number of paths into which the original list is recommended to be decomposed. The formula that relates the number of paths $M$ to the number of processors $P$ and the data size $N$ is then given by:

$$M \leq P \times \left(\frac{\log N}{2}\right)^{-2}.$$

If we compare the experimental results of the jump searching merge algorithm with the results of the binary merge algorithm presented in Tables 6.16 and 6.14 respectively, we notice that the jump merge algorithm requires less running-time than the binary merge algorithm. Therefore, a slightly higher speed-up is achieved in the jump merge algorithm. In addition, the total speed-ups in Table 6.16 are greater than those of the binary merge algorithm in Table 6.14. Therefore, the two levels simple jump merge algorithm is faster than the binary merge algorithm and the parallel implementation also achieves better results in this jump merging algorithm.

**The Two Level Fixed Jump Merging Method**

In this method, the merge proceeds as in the two level simple jump merging except that the jumps are of size $N^{1/3}$ and $N^{2/3}$. We call this method Prog 6.8. Since this method requires $\frac{3}{2}N^{1/3}$ comparisons (see equation (6.3.5.5)), therefore, we can proceed with the analysis of this method when it is applied on $M$ sorted subsets in the same manner as discussed in the
<table>
<thead>
<tr>
<th>N</th>
<th>M</th>
<th>P</th>
<th>$S_{total}$</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1024)</td>
<td>4</td>
<td>2</td>
<td>1.99</td>
<td>0.997</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2.98</td>
<td>0.992</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>3.94</td>
<td>0.986</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>1.99</td>
<td>0.99</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2.97</td>
<td>0.989</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3.92</td>
<td>0.979</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4.84</td>
<td>0.969</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5.75</td>
<td>0.959</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>6.65</td>
<td>0.949</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7.52</td>
<td>0.940</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>2.0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2.97</td>
<td>0.989</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3.87</td>
<td>0.969</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4.75</td>
<td>0.95</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5.58</td>
<td>0.93</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>6.39</td>
<td>0.91</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7.16</td>
<td>0.895</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 6.18:** The theoretical values of $S_{total}$ of Prog 6.7
previous section. Here,
\[ T(1) = \frac{3}{2} i^{1/3}, \text{ then,} \]
\[ W_{i+1} < W_i \frac{1}{2^{1/3}} \cdot \frac{3}{2} (1+i)^{1/3} \]
which implies,
\[ W_i < W_2 \frac{2}{2^{1/3}} \cdot \frac{(i)^{1/3}}{i!} \]
Therefore,
\[ T(s) < s \left( \frac{s-1}{s} \right)^{s} + s.s! \left( \frac{s-1}{s} \right)^{s} \cdot \frac{2}{2^{1/3}} W_2 \sum_{i=2}^{s} \frac{1}{i!} \]
Since,
\[ W_2 = \frac{T(2)}{2!(s-2)!((s-1)^2)} = \frac{3}{2} \cdot 2^{1/3} \]
\[ (\frac{s-1}{s}) \lesssim \frac{1}{e} \]
and \[ \sum_{i=2}^{s} \frac{1}{i!} < \sum_{i=2}^{s} \frac{1}{i!} = \sum_{i=1}^{\infty} \frac{1}{i!} < e. \]
Therefore,
\[ T(s) < s \left( \frac{s-1}{s} \right)^{s} \lesssim 3 \left( \frac{s-1}{s} \right) < 2s \left( \frac{s-1}{s} \right) \]
Then, the sequential complexity \( T_{IM} \) and the parallel complexity \( T_{PM} \) of Prog 6.8 are given by:
\[ T_{IM} < N \log M, \]
and
\[ T_{PM} < \frac{N}{p} \log \left( \frac{M}{P} \right) + \frac{2N}{P} (P-1) + \log M. \]
Hence, the merge speed-up is given by:
\[ S_{merge} < \frac{N \log M}{\left( \frac{N}{p} \log \left( \frac{M}{P} \right) + 2N/P (P-1) + \log M \right)} \]
This gives that \( S_{merge} \) is of \( O \left( \frac{p \log M}{\log \left( \frac{M}{P} \right) + 2P-2} \right) \) which is as obtained from the first method. The theoretical values of this \( S_{merge} \) are listed in Table 6.19.

As Prog 6.8 includes the bubble sort and the jump merge algorithms, therefore the total speed-up of Prog 6.8 is given by:
The experimental results of Prog 6.8 are listed in Table 6.20, and the theoretical values of $S_{\text{total}}$ of inequality (6.3.5.13) are listed in Table 6.21. From Table 6.21, we can derive a formula for the maximum number of processors one can use to obtain approximately a linear speed-up. This formula is given by:

$$P \leq \left\lfloor \frac{\log (N/M)}{2} \right\rfloor,$$

where $N$ is the data size and $M$ is the number of paths. This limit equals that obtained from Prog 6.7. Also, the maximum number of paths is similar to that of Prog 6.7, i.e.,

$$M \leq P \times \left\lfloor \frac{\log N}{2} \right\rfloor - 2.$$

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
$N$ & $P$ & $M$ & $S_{\text{merge}}$ & Theory & Experiment \\
\hline
\multirow{5}{*}{(1024)} & 4 & 4 & 1.33 & 1.34 \\
& & 8 & 1.71 & 1.73 \\
& & 16 & 1.99 & 2.02 \\
& & 32 & 2.22 & 2.25 \\
& & 64 & 2.39 & 2.43 \\
\hline
\multirow{5}{*}{2} & 4 & 4 & 1.33 & 1.34 \\
& & 8 & 1.50 & 1.51 \\
& & 16 & 1.60 & 1.61 \\
& & 32 & 1.66 & 1.67 \\
& & 64 & 1.71 & 1.72 \\
\hline
\end{tabular}
\caption{The theoretical values of $S_{\text{merge}}$ of the formula (6.3.5.12)}
\end{table}
<table>
<thead>
<tr>
<th>No. of processors</th>
<th>No. of paths</th>
<th>Time for sorting</th>
<th>Time for merging</th>
<th>Total time</th>
<th>Speed-up for sorting</th>
<th>Speed-up for merging</th>
<th>Total Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>88.050</td>
<td>4.970</td>
<td>93.020</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>47.630</td>
<td>3.720</td>
<td>51.350</td>
<td>1.85</td>
<td>1.34</td>
<td>1.81</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>23.330</td>
<td>3.720</td>
<td>27.050</td>
<td>3.77</td>
<td>1.34</td>
<td>3.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td>43.030</td>
<td>7.650</td>
<td>50.680</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23.280</td>
<td>5.070</td>
<td>28.350</td>
<td>1.85</td>
<td>1.51</td>
<td>1.79</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16.910</td>
<td>5.070</td>
<td>21.980</td>
<td>2.55</td>
<td>1.51</td>
<td>2.31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11.640</td>
<td>4.430</td>
<td>16.070</td>
<td>3.70</td>
<td>1.73</td>
<td>3.15</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>20.970</td>
<td>10.230</td>
<td>31.200</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>11.260</td>
<td>6.370</td>
<td>17.630</td>
<td>1.86</td>
<td>1.61</td>
<td>1.77</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>8.680</td>
<td>6.030</td>
<td>14.710</td>
<td>2.41</td>
<td>1.70</td>
<td>2.12</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5.890</td>
<td>5.070</td>
<td>10.960</td>
<td>3.56</td>
<td>2.02</td>
<td>2.85</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>10.250</td>
<td>13.00</td>
<td>23.250</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>5.670</td>
<td>7.770</td>
<td>13.440</td>
<td>1.81</td>
<td>1.67</td>
<td>1.73</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>4.220</td>
<td>7.080</td>
<td>11.300</td>
<td>2.43</td>
<td>1.84</td>
<td>2.06</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2.950</td>
<td>5.780</td>
<td>8.730</td>
<td>3.47</td>
<td>2.25</td>
<td>2.66</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
<td>4.850</td>
<td>15.620</td>
<td>20.470</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2.660</td>
<td>9.090</td>
<td>11.750</td>
<td>1.82</td>
<td>1.72</td>
<td>1.74</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1.990</td>
<td>7.950</td>
<td>9.940</td>
<td>2.45</td>
<td>1.96</td>
<td>2.06</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1.430</td>
<td>6.430</td>
<td>7.860</td>
<td>3.39</td>
<td>2.43</td>
<td>2.60</td>
</tr>
</tbody>
</table>

*TABLE 6.20:* The experimental results of Prog 6.8: The Two Level Fixed Jump Merge and Bubble Sort algorithms
TABLE 6.21: The theoretical values of the total speed-up of Prog 6.8

Another method using the jump merging strategy has been developed. This method which we call Prog 6.9 has two levels of jumps where the first level's jump size is \((S/3)\), where \(S\) is the size of the subset. When the required position to insert an element into a subset is not found another jump of the size \((S/9)\) is done and similar to the previous method a simple search is performed on the remaining segment of the subset until that position is found. The experimental results of this method are shown in
Table 6.22, where the bubble sort algorithm is implemented to sort the $M$ subsets.

The results in Table 6.22 explain that this method requires less time than the method of Prog 6.8 but the relative speed-up is approximately similar to that of Prog 6.8.

The small timing results of this method is due to the fact that the jump sizes are greater than that of the first two methods. This means that if successive comparisons are required after the second jump a segment of small size is searched which means a small number of comparisons is required in the sequential search part. On the other hand, the two level fixed jump method, for example, requires more comparisons in the sequential search part, since the segment left is relatively larger than that of the third method. These reasons, actually, explain the difference in the run-time of these two methods.

The total speed-ups in the first two methods are the same, but the total speed-up in the third method (of jump sizes as $s/3$ and $s/9$, where $s$ is the size of the subsets) is slightly greater than that of the other two methods. This difference is obvious in the case $P=4$ for all cases of $M$.

In order to measure the merge speed-up and the total speed-up theoretically, we need to estimate the total number of comparisons for this method. However, if we assume that the jump sizes are $N/3$ and $N/9$, then by substituting these values in equation (6.3.5.3) we obtain the total complexity for the data size $N$. Thus,

$$C(N) = \frac{1}{3} \frac{N}{N/3} + \frac{1}{3} \frac{N/3}{N/9} + \frac{1}{3} \left( \frac{N}{9} - 1 \right)$$
$$= \frac{2}{9} + \frac{5}{2} \cdot \frac{N}{18}.$$  
\[ (6.3.5.14) \]

Now, we proceed with the analysis of this method as in the previous methods. Thus,

$$T(s) = \sum_{i=1}^{s} \binom{s}{i} (s-1)^{(s-1)} \frac{1}{s} s^{-1} T(i).$$
<table>
<thead>
<tr>
<th>No. of processors</th>
<th>No. of paths</th>
<th>Time for sorting</th>
<th>Time for merging</th>
<th>Total time</th>
<th>Speed-up for sorting</th>
<th>Speed-up for merging</th>
<th>Total Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>88.050</td>
<td>3.690</td>
<td>91.740</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>47.630</td>
<td>2.790</td>
<td>50.420</td>
<td>1.85</td>
<td>1.32</td>
<td>1.82</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>23.330</td>
<td>2.790</td>
<td>26.120</td>
<td>3.77</td>
<td>1.32</td>
<td>3.51</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>43.030</td>
<td>5.730</td>
<td>48.760</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>23.280</td>
<td>3.840</td>
<td>27.120</td>
<td>1.85</td>
<td>1.49</td>
<td>1.80</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>16.910</td>
<td>3.840</td>
<td>20.750</td>
<td>2.55</td>
<td>1.49</td>
<td>2.35</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>11.640</td>
<td>3.390</td>
<td>15.030</td>
<td>3.70</td>
<td>1.69</td>
<td>3.24</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>20.970</td>
<td>7.770</td>
<td>28.740</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>11.260</td>
<td>4.860</td>
<td>16.120</td>
<td>1.86</td>
<td>1.60</td>
<td>1.78</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>8.680</td>
<td>4.830</td>
<td>13.510</td>
<td>2.41</td>
<td>1.61</td>
<td>2.13</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5.890</td>
<td>3.910</td>
<td>9.800</td>
<td>3.56</td>
<td>1.98</td>
<td>2.93</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>10.250</td>
<td>9.770</td>
<td>20.020</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>5.670</td>
<td>5.850</td>
<td>11.520</td>
<td>1.81</td>
<td>1.67</td>
<td>1.74</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>4.220</td>
<td>5.340</td>
<td>9.560</td>
<td>2.43</td>
<td>1.83</td>
<td>2.09</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2.950</td>
<td>4.370</td>
<td>7.320</td>
<td>3.47</td>
<td>2.24</td>
<td>2.73</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
<td>4.850</td>
<td>12.630</td>
<td>17.480</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2.660</td>
<td>7.310</td>
<td>9.970</td>
<td>1.82</td>
<td>1.73</td>
<td>1.75</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1.990</td>
<td>6.430</td>
<td>8.420</td>
<td>2.45</td>
<td>1.96</td>
<td>2.08</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1.430</td>
<td>5.250</td>
<td>6.680</td>
<td>3.39</td>
<td>2.41</td>
<td>2.62</td>
</tr>
</tbody>
</table>

TABLE 6.22: The experimental results of Prog 6.9: The Two Level Jump Merge of sizes (N/M)/3 and (N/M)/9, N=1024
Let \( T(1) = a \), then,

\[
T(s) = a \sum_{i=2}^{s} \frac{(s-1)!}{i!(s-i)!} s^{i-1}.
\]

If we let,

\[
W_i = \frac{T(i)}{i!(s-i)!(s-1)!},
\]

then,

\[
W_{i+1} < W_i \frac{1}{i+1} \frac{T(i+1)}{T(i)}.
\]

Since, \( T(i) \leq \frac{i}{18} + \frac{5}{2} \), then,

\[
W_{i+1} < W_i \frac{1}{i+1} \cdot \frac{\left(\frac{i+1}{18} + \frac{5}{2}\right)}{\left(\frac{i}{18} + \frac{5}{2}\right)} < W_i \frac{1}{i+1} \frac{(i+1)+(45)}{(i+1)+45}.
\]

Thus,

\[
W_1 < W_2 \cdot \frac{2}{47} \cdot \frac{(i+45)}{i!}.
\]

Therefore,

\[
T(s) < \frac{s(s-1)}{s} \sum_{i=2}^{s} \frac{(s-1)^i}{i!} s^{i-1} \cdot \frac{2}{47} W_2 \sum_{i=2}^{\infty} \frac{(i+45)}{i!}.
\]

\[
< s \left(\frac{s}{s-1}\right) \frac{1}{e} + s! \frac{1}{e} \cdot \frac{2}{47} \sum_{i=2}^{\infty} \frac{(i+45)}{i!}.
\]

\[
< s \left(\frac{s}{s-1}\right) \frac{1}{e} + \frac{1}{9e} s \left(\frac{s}{s-1}\right) (e + 45e)
\]

\[
< s \left(\frac{s}{s-1}\right) \left(\frac{1}{e} + \frac{46}{9}\right) < 5 s \left(\frac{s}{s-1}\right).
\]

Hence, the total complexity for the uni-processor implementation is:

\[
T_{1M} < \frac{5}{2} N \log M, \tag{6.3.5.15}
\]

and the total complexity for the parallel implementation is given by:

\[
T_{PM} < \frac{5N}{2P} \log \left(\frac{M}{P}\right) + \frac{5N}{P} (P-1) + \log M. \tag{6.3.5.16}
\]

The merge speed-up is then given by:

\[
S_{\text{merge}} < \frac{\frac{5}{2} N \log M}{\frac{5N}{2P} \log \left(\frac{M}{P}\right) + \frac{5N}{P} (P-1) + \log M}
\]

\[
< \frac{P \log M - \frac{\log M}{T_{PM}}}{(\log (\frac{M}{P}) + 2P - 2)} \tag{6.3.5.17}
\]

The above \( S_{\text{merge}} \) is also of \( O\left(\frac{P \log M}{\log (\frac{M}{P}) + 2P - 2}\right) \) which is similar to the other two methods. This is also indicated from the theoretical results obtained from inequality (6.3.5.17) and listed in Table 6.23.
The total speed-up of Prog 6.9 is given by:

\[
S_{\text{total}} < \frac{(3N^2 - \frac{1}{4}N\ln(N)}{4M} - \frac{3N}{4} + \frac{5N}{2} \log M)
\]

\[
\frac{(3N^2 - \frac{1}{4}N\ln(N)}{4MP} - \frac{3N}{4P} + \frac{5N}{2P} \log(M)}{P} + \frac{5N}{P(P-1)} + \log M)
\]

Some theoretical values of \(S_{\text{total}}\) are given in Table 6.24.

<table>
<thead>
<tr>
<th>N</th>
<th>P</th>
<th>M</th>
<th>S_{merge}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Theory</td>
</tr>
<tr>
<td>(1024)</td>
<td>2</td>
<td>4</td>
<td>1.33</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.50</td>
<td>1.49</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.60</td>
<td>1.60</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.66</td>
<td>1.67</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1.71</td>
<td>1.73</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1.33</td>
<td>1.32</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.71</td>
<td>1.69</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2.0</td>
<td>1.98</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>2.22</td>
<td>2.24</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>2.40</td>
<td>2.41</td>
</tr>
</tbody>
</table>

TABLE 6.23: The theoretical values of \(S_{\text{merge}}\) of Prog 6.9
From Table 6.24, we notice that the efficiency of the algorithm in Prog 6.9 decreases as P increases for a fixed value of M. However, the maximum number of processors P one can use and the maximum number of paths M of this algorithm can then be predicted. These values are respectively given by:

\[ P \leq \left\lfloor \frac{\log(N/M)}{2} \right\rfloor + 1 \] processors,

and

\[ M \leq P \left( \left\lfloor \frac{\log N}{2} \right\rfloor - 3 \right) \] paths.
It can be noticed that the above two quantities are less than that of the other two jump methods.

Although, all the three jump merge methods and the binary merge method are implemented in the same way (i.e. all require log M steps to complete the merge), they differ slightly in their performance analysis which is presented in Table 6.25. This difference is due to the different timing results obtained from the experiments which affect the percentage values.

The static and the contention for parallel paths are taken from the experimental results where the number of processors P=4. The parallel path access rate is predicted from the program and, it is obvious from Table 6.25 that these predicted results are in agreement with the results obtained from the 4-processor run.

However, the access rate to a parallel path for the three jump merge methods is taken as one access to a path per the complexity (operations) required in this path. As in the 2-way merge algorithm discussed earlier, we need to take the average of the sizes of the subsets to measure the complexity per path. Since we have log M steps in all the methods, therefore, in the two level simple jump for example, the complexity to merge two subsets of size (N/M) is \((\frac{1}{2}(N/M)^{\frac{1}{2}} + (N/M)^{\frac{1}{2}}) \cdot N/M\) comparisons. Since the size of the subset is doubled in each successive step, therefore, we choose log M to be multiplied by the above number of comparisons to obtain the total expected complexity. Similarly, for the other jump merge methods where their complexity should be taken from their tasks presented earlier.

The losses measured from these predicted quantities are obtained from the measurements on the NEPTUNE system presented in Table 5.10 (Chapter 5). These losses are given as a percentage for different values of M when N=1024.
The contention losses in all the methods decreases as \( M \) increases. This is because when \( M \gg P \), then most of the time the \( P \) processors are busy except in the few last steps of the algorithms where \( P \) is halved.

The contention losses in the third method in Table 6.25 are larger than the other methods and this is due to the smaller timing results that this method achieved on which the losses are taken as a percentage. The static shared data losses are predicted as two accesses per one operation and this results in -0.2% loss which is what was obtained in all the jump methods as well as the other merge methods described in the previous section.

In conclusion, all of the three jump merge methods present smaller timing results than that of the binary merge method and the method of the jump sizes \((S/3)\) and \((S/9)\) has the smallest timing results amongst the other methods. From the speed-up point of view, all of these methods have achieved the same speed-up ratios which are adequately presented by the predicted analytical results.
<table>
<thead>
<tr>
<th>Method</th>
<th>Paths</th>
<th>Shared Data</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>access rate</td>
<td>loss</td>
<td></td>
<td>access rate</td>
<td>loss</td>
<td>static</td>
<td>contention</td>
</tr>
<tr>
<td>Two level simple jump</td>
<td>4</td>
<td>2:1 flop</td>
<td>0.2%</td>
<td></td>
<td>1:($\frac{N}{M}^1 + \frac{N}{M}^1$)</td>
<td>0.02%</td>
<td>0.02%</td>
<td>2.61%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.033%</td>
<td>0.039%</td>
<td>2.23%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.065%</td>
<td>0.06%</td>
<td>1.9%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.14%</td>
<td>0.11%</td>
<td>1.71%</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.29%</td>
<td>0.20%</td>
<td>1.53%</td>
</tr>
<tr>
<td>Two level fixed jump</td>
<td>4</td>
<td>2:1 flop</td>
<td>0.2%</td>
<td></td>
<td>1:($\frac{3N}{2M}^1 + \frac{N}{M}^1$)</td>
<td>0.02%</td>
<td>0.03%</td>
<td>2.22%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.039%</td>
<td>0.036%</td>
<td>1.94%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.07%</td>
<td>0.06%</td>
<td>1.72%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.15%</td>
<td>0.11%</td>
<td>1.53%</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.3%</td>
<td>0.20%</td>
<td>1.53%</td>
</tr>
<tr>
<td>Two level jumps(S, S/3, S/9)</td>
<td>4</td>
<td>2:1 flop</td>
<td>0.2%</td>
<td></td>
<td>1:($\frac{N}{18}^1 + \frac{5}{2}$)</td>
<td>0.014%</td>
<td>0.02%</td>
<td>3.5%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.031%</td>
<td>0.047%</td>
<td>2.9%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.074%</td>
<td>0.08%</td>
<td>2.52%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.17%</td>
<td>0.15%</td>
<td>2.27%</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.3%</td>
<td>0.24%</td>
<td>1.87%</td>
</tr>
<tr>
<td>Binary merge</td>
<td>4</td>
<td>2:1 flop</td>
<td>0.2%</td>
<td></td>
<td>1:($\frac{N}{M}^{\log N}{M}$)</td>
<td>0.014%</td>
<td>0.01%</td>
<td>1.27%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.021%</td>
<td>0.01%</td>
<td>1.11%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.021%</td>
<td>0.01%</td>
<td>1.11%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.071%</td>
<td>0.01%</td>
<td>1.01%</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>0.2%</td>
<td></td>
<td>(\frac{N}{M} \log M) flops</td>
<td>0.15%</td>
<td>0.12%</td>
<td>0.88%</td>
</tr>
</tbody>
</table>

TABLE 6.25: The performance measurements of the Jump merge and Binary merge methods
6.4 CONCLUSIONS

In this chapter, several merge algorithms were presented. First, three types of algorithms were compared from the usage of the processors point of view. However, in the odd-even method the processors were fully used and none of them are idle especially for $M \gg P$. Meanwhile, in the 2-way merge, whenever $M$ was greater than $P$, in the later steps of the algorithm, some of the processors become idle and always in the last step of the algorithm only one processor is active and the rest remain idle. For this reason, these methods have different parallel path overheads as shown in Table G.13. However, the static parallel path overhead and the contention losses were obtained from the results of the four processors run. Thus, in the 2-way merge, the static overhead loss slightly increases when $M$ increases, whereas, in the odd-even merge, the increase was more rapid and the overhead was greater than that of the 2-way merge when a particular value of $M$ was considered. This is because the odd-even method implements $M$ steps which are performed sequentially, i.e. synchronisations are required after each step within which parallelism exists. However, in the new merge algorithm, where $(\log M + 1)$ steps are required, the static parallel path loss was less than that of the 2-way merge since in the new merge, all the processors are fully used in the first step then the number of processors is halved in successive steps. Therefore, the new merge method has the smallest static parallel path losses amongst all three methods.

On the other hand, the contention losses arising from the processors being kept waiting for a path to be executed decrease as $M$ increases in the three methods. However, the decrease is obvious in the odd-even method since in this method all the processors are busy most of the time in evaluating the paths. Once again, the new merge method has the less contention losses than the 2-way method for the same reasons as discussed earlier.
The predicted parallel path access losses were in agreement with what we obtained from the experiments. The parallel path access rate is measured from the program itself. This measurement indicates how many accesses to a path per the number of the main floating point operations required in the algorithm. Other operations involved in the program are not included.

The main operations in the merge algorithms are the total number of comparisons. Since the merge of each two subsets of size $N/M$ each, requires $(2N/M -1)$ comparisons, therefore the odd-even merge method has one access to a path per $(2N/M -1)$ operations. Whilst, in the 2-way merge method, the size of the subsets is doubled in each step of the algorithm where the access to a path is made. In this case, we take the average of the sizes of the log $M$ steps of the algorithm and this gives a result of one access to a path per $(2N/M -1)\log M$ operations. On the other hand, in the new merge method, since $(\log M+1)$ steps are required, where in each step, the 2-way merge algorithm is implemented sequentially to merge the subsets whose sizes doubled in consequent steps, therefore the average number of comparisons per path is given by $M \log M \frac{2N}{M}^2 (1 - 1)$ $(\log M + 1)$. The term $\frac{2N}{M}^2$ is given here since the size of the subsets at the beginning of the algorithm is $\frac{N}{2}$, thus the total number of comparisons to merge two subsets is $\frac{2N}{M}^2 (1 - 1) \log M$ using the 2-way merge multiplied by $M$ (the number of the subsets). These predictions give approximate results to those given by the experiments.

From the shared data losses point of view, all the methods have made 2 accesses to the shared data per one operation and this results in a loss equal to approximately 0.2%. These are predicted from the program. Since there was no hardware for the floating point operations, the experimental shared data loss was not measured.
For the other two types of algorithms, the binary merge and the jump-merge algorithms with its variants which were implemented on the NEPTUNE system, then all the variants of the jump merge algorithms present less timing results than the binary merge although they achieve approximately equal speed-up ratios. Since these four methods are implemented in the same manner, i.e. they all require log M steps to merge M subsets, then they all require the same number of paths to evaluate. However, the static losses of the parallel paths when taken as a percentage of the timing results, yield slightly different results. Also, the static shared data loss is 0.2% for the four methods.

Finally, we conclude that the odd-even merge method exploited the maximum parallelism amongst all the other methods although the 2-way merge method presented the least timing results amongst the others. The jump merge methods have proved to achieve better timing results than the binary merge algorithm. In general, the jump search algorithm, can be implemented in index searches (Maruyama and Smith, [1977]) where it was proved that jump searching can be more efficient than binary searching when each node of the tree structured index has certain properties.
CHAPTER 7

SORTING METHODS BY PARALLEL DISTRIBUTIVE

PARTITIONING ALGORITHM
7.1 **INTRODUCTION**

In Chapter 6, we mentioned that since the early days of computer science two fundamentally different approaches have been taken in the design of sorting algorithms: sorting by comparison and sorting by distribution (Flores [1969], Martin [1971], Knuth [1973]). However, in that chapter, we discussed the problem of sorting two or more sets of data where some sorting methods based upon comparisons had been described and compared.

In sorting by distribution, the set of elements to be sorted is partitioned into a number of subsets, often called 'buckets'. The contents of two consecutive buckets are such that all the elements in the first one are guaranteed to precede all the elements in the second. Each bucket is then sorted using the same principle (Martin [1971]). The Distributive Partitioning Sorting algorithm (DPS) of Dobosiewicz [1978] is an example of this approach. His approach required a median calculation of the set of elements which requires a large number of comparisons.

However, the DPS algorithm has received a good deal of attention in the literature recently. For example, Huits and Kumar [1979] have improved the above algorithm by using a straight insertion sort technique to sort a bucket of small size. The evidence so far indicates that this approach is in many cases the fastest known sorting technique and, in fact, it was the first algorithm to achieve a linear expected run-time (i.e. of $O(N)$) for uniformly distributed inputs, while still having a worst-case time-complexity of $O(N \log N)$, where $N$ is the size of the set to be sorted.

Other experiments by Kowalik and Yoo [1981] show that this algorithm is roughly 35% faster than an efficient version of Quicksort proposed by Sedgewick [1978]. These experiments which avoided the median calculation
were carried out on an Amdahl V/6 computer at Washington University for both uniform and normally distributed inputs. Also, Meijer and Akl [1980] have presented a new hybrid sorting algorithm which sorts partly by distribution and partly by comparison. Their algorithm has two important features which are: firstly unlike DPS it is quite simple to state, to program and to analyse; secondly, the property of the expected time-complexity is not limited to uniformly distributed inputs, and the same behaviour is exhibited for a large family of probability distributions.

For a slightly different point of view, Nat [1980] tried to avoid the median calculation of Dobosiewicz [1978] algorithm, which requires on average $\frac{3}{2}N + O(N^\frac{1}{2})$ comparisons (Floyd and Rivest [1975]), by another method that requires fewer comparisons. This last approach includes a repeated partitioning procedure which yields two sorted sets of buckets and a merge procedure which merges two sets to produce the final sorted set. However, the merge procedure requires $(N-1)$ comparisons which is, of course, less than that obtained from the median calculation, although the same expected time-complexity was obtained. Finally, an implementation by Allison and Noga [1982] which requires one distributive pass followed by two comparison passes was suggested. The results of this implementation when run on an IBM 3032 indicated that it is a very fast sorting method which is about 53-73% of the Quicksort timing results.

Thus, in all the above-mentioned methods for the DPS algorithm, the expected run-time complexity is of $O(N)$ which is better than that of the Quicksort which is of $O(N \log N)$. The only drawback of the DPS algorithm is the space requirement which is much more than that of the Quicksort algorithm.

In our experiments, we implemented the Quicksort and four versions of the DPS algorithm on the NEPTUNE system and compared their results and overhead measurements.
7.2 **PARALLEL QUICKSORT ALGORITHM**

The Quicksort algorithm, which was first discovered by Hoare [1961] and [1962], is considered to be the most efficient general-purpose sorting algorithm as both empirical and analytic studies have proved (Loeser [1974] and Knuth [1973]). Quicksort is a comparative, interchange sorting algorithm and also is an example of a distributive sort since it is based on partitioning the original set into smaller subsets.

Sedgewick [1975] has proposed the most efficient improvements to Quicksort and also presented the history and a complete survey of the many variants of the Quicksort that have been proposed. Also Sedgewick [1978] and [1977] presented and analysed the proposed algorithms which introduced linear insertion to sort the subsets of smaller size and found that the choice of the median of three elements as a partitioning element is better than randomly choosing that element. Thus, these improvements raised the efficiency and slightly reduced the running-time of the algorithm.

Our implementation of the parallel Quicksort is an extension to the algorithm proposed by Barlow [1977]. Our algorithm is based on finding the median of three elements as a partitioning element. The three elements are the first, middle, and last elements of the set. When the partitioning element, say $V$, is found the original set is partitioned into two subsets $S_1$ and $S_2$ such that all the elements of $S_1$ are less than $V$ and all elements of $S_2$ are greater than $V$. The two subsets $S_1$ and $S_2$ are then sorted by the same procedure if they are sufficiently large. Otherwise, a linear insertion sort procedure is used to sort the subsets. It is found that the size of the subset to be sorted by the linear insertion sort is less than $M$, where $M$ is a constant (Sedgewick [1975]). The experiments showed that the best value of $M$ is between 5 and 20.
For the partitioning procedure, two queue data structures are used to hold the pointers of the start and the end of the resultant subset in First-In-First-Out concept. Therefore, in order to sort the subset, its corresponding start and end pointers are popped from the queue. If this subset is larger than M then it is partitioned and the pointers of the resulting subsets are pushed onto the queue in a similar fashion as before. Otherwise, the subset is sorted by the linear insertion sort procedure and the results are placed in the resultant output vector which is the same as the input vector but its elements are sorted. The input vector of the elements to be sorted and the two queues are kept in the shared memory in order to be accessed by all the processors. This algorithm is implemented as Prog 7.1 which is shown as Program A.10 in Appendix A.

It is important to know in advance the maximum number of the locations used by the queue, in order to ensure that the number of sublists which are postponed at any time never exceeds the logarithm of base 2 of the number of elements to be sorted. However, it is very important to economise on the storage of the sublists' details, since this number is proportional to N, the number of elements being sorted.

In fact, the nature of this algorithm makes it highly recommended to be implemented on a parallel MIMD computer. Since the partitioning procedure is continuously proceeding to produce numbers of subsets of smaller sizes of the current considered set, therefore, these subsets can be sorted simultaneously as long as many processors are available.

In the parallel implementation of Prog 7.1, only one path is generated first to partition the original set into two subsets which are placed in the queue. When the queue is not empty, the process is repeated on these two subsets, i.e. two paths are generated to yield four subsets. The partitioning of the two subsets is performed simultaneously by two processors.
Next, the four subsets are partitioned by four processors to yield eight subsets and so on until there are no further subsets to partition. This process is represented by the top-bottom tree diagram in Figure 7.1. If we assume that the subsets produced by the partitioning procedure are of equal size then, if the size of the original set is $N$, the size of the subsets in the subsequent steps of the algorithm will be $N/2$, $N/4$ and so on.

The above algorithm illustrates that the number of parallel paths generated in each step depends on the number of subsets produced by the previous step which are kept in the queue. After some steps, the number of the generated paths will be greater than the number of the available processors. In this case, the first $P$ subsets, where $P$ is the number of available processors, are processed and the others will be held in a queue to wait for a free processor. The step is completed when all subsets
have been partitioned. This means that the algorithm is synchronised at each step which makes some of the paths to wait for the slower ones.

One should notice that when the new subsets produced from the partitioning procedure are placed on the queue, only one processor at a time must modify this queue information. This, however, can be achieved by using a critical section mechanism to ensure that only one processor at a time is adding information to the queue.

Experiments on this algorithm have been carried out on the NEPTUNE system where the set of the elements to be sorted is randomly generated by using a random generator in the interval \([0,1]\). These experiments were performed for different values of \(M\), on which the linear insertion sort procedure is activated. The results of such experiments are listed in Table 7.1. These results showed that the usage of only 2 processors is better than that of 4 processors. This is indicated by the speed-up factors obtained from the two cases which is clarified by the following analysis. One can notice from Table 7.1 that values of \(M\) between 5-25 are not very crucial to the speed-up values, although the running-time slightly decreases when \(M\) increases.

7.2.1 The Quicksort Analysis

The sequential version of the Quicksort algorithm has been fully analysed by Knuth [1973] when the partitioning element is chosen at random and by Sedgewick [1977] when the partitioning element is chosen as the median of three elements. In their analysis, many quantities which are relevant to the algorithmic strategy were found. These are: the partitioning stages, number of exchanges during partitioning, number of comparisons during partitioning, number of insertions, and the number of keys moved during insertion.

As with all our sorting and merging algorithms, if we consider the
<table>
<thead>
<tr>
<th>N</th>
<th>No. of Processors</th>
<th>M</th>
<th>Time (sec.)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1024)</td>
<td>1</td>
<td>5</td>
<td>10.900</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>6.810</td>
<td>1.60</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>5.260</td>
<td>2.07</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>4.580</td>
<td>2.37</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>7</td>
<td>10.430</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>6.640</td>
<td>1.57</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>5.490</td>
<td>1.90</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>4.700</td>
<td>2.22</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>12</td>
<td>9.010</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>5.930</td>
<td>1.52</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>4.660</td>
<td>1.93</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>4.210</td>
<td>2.14</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>15</td>
<td>8.180</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>5.350</td>
<td>1.53</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>4.320</td>
<td>1.89</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>3.850</td>
<td>2.12</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>20</td>
<td>8.330</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>5.330</td>
<td>1.56</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>4.370</td>
<td>1.91</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>3.870</td>
<td>2.15</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>25</td>
<td>8.890</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>5.650</td>
<td>1.57</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>4.680</td>
<td>1.90</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>4.180</td>
<td>2.13</td>
</tr>
</tbody>
</table>

**TABLE 7.1**: Experimental results of Prog 7.1: The Parallel Quicksort algorithm
time complexity of the algorithm to be measured with respect to the number of comparisons only, then we can find the total sequential time-complexity by following Sedgewick's analysis of the median of three elements as follows:

Let $N$ be the data size and $M$ the constant by which the linear insertion sort is used, then the probability that the $k$th smallest element is the partitioning element is given by:

$\frac{(N-k)(k-1)}{\binom{N}{3}}.$

Now, if we let $F_N$ denote the average value of the number of comparisons when a random arrangement of $N$ items is sorted, and $f_N$ denotes the average value of the comparisons for the first partitioning step, then,

$$F_N = f_N + \sum_{k=1}^{N-1} \binom{N-k}{3} \left( F_{k-1} + F_{N-k} \right),$$

for $N > M^3$. (7.2.1.1)

The quantities $F_{k-1}$ and $F_{N-k}$ in equation (7.2.1.1) represent respectively, the average number of comparisons for the left subset and for the right subset to the partitioning element. The solution of equation (7.2.1.1), which is obtained by using some generating functions and with the assumption that $f_N = N+1$, is given by Sedgewick [1977] as follows:

$$F_N = \frac{12}{7} (N+1) \log_e \left( \frac{N+1}{M+2} \right) + \frac{37}{49}(N+1) - \frac{24}{7} \frac{N+1}{(N+2)} + 2 + O(N^{-6})$$

Therefore, the sequential time-complexity $T_1$ for Prog 7.1 is given by:

$$T_1 = \frac{12}{7} (N+1) \log_e \left( \frac{N+1}{M+2} \right).$$

(7.2.1.3)

Now, to obtain the total time-complexity $T_p$ for Prog 7.1, we define $T_p$ as follows:

$$T_p = t_1 + t_2,$$

(7.2.1.4)

$^\dagger$ where $\binom{n}{r}$ represents $\frac{n(n-1) \ldots (n-r+1)}{r!}$.
where $t_1$ is the time taken in the partitioning stage where all the available processors $P$ (of power 2) are employed concurrently and $t_2$ is the time taken in all the steps of the partitioning procedure before the stage defined in $t_1$. For example, when $P=4$ we will have to partition the set of $N$ elements and the two subsets of $N/2$ elements before the 4 processors are fully used. However, in order to reach the stage of $t_1$, log $P$ previous steps are processed. Therefore, the time spent in this stage is calculated by the following formula:

$$t_1 = \frac{q_1-q_2}{P}$$

where $q_1$ = sequential time of the Quicksort,
$q_2$ = sequential time of $t_2$, and
$P$ = number of available processors.

$q_1$ which is equal to $T_1$ has been found earlier. The sequential time of $t_2$ (i.e. $q_2$) can be easily found. Thus, since the total work in each level of the tree in Figure 7.1 is approximately $(N+1)$, therefore,

$$q_2 = (N+1) \log P.$$  \hspace{1cm} (7.2.1.5)

Hence,

$$t_1 = \frac{12(N+1)\log \frac{(N+1)}{N+2} - (N+1) \log P}{P}$$ \hspace{1cm} (7.2.1.6)

Next, we calculate $t_2$ as follows:

In the first partitioning step $(N+1)$ comparisons are required. The second step employs two processors since two sublists of size $N/2$ are to be partitioned. Therefore, $\frac{N}{2} + 1$ comparisons are required. This process continues until the $(\log P-1)$th step in which $\left(\frac{N}{2(\log P-1)} + 1\right)$ comparisons are implemented.

By summing up the comparisons of all the $(\log P - 1)$ steps, we obtain $t_2$. Thus,

$$t_2 = N+1 + \frac{N}{2} + 1 + \ldots + \frac{N}{2(\log P-1)} + 1$$

$$= 2N\left(1 - \frac{1}{2\log P}\right) + \log P - 1$$
\[
359
\]

\[= \frac{2N}{P} (P-1) + \log P - 1. \quad (7.2.1.7)\]

By adding equations (7.2.1.6) and (7.2.1.7), we obtain:

\[
T'_P = \frac{(N+1)\left[\frac{12}{7} \log \left(\frac{N+1}{M+2}\right) - \log P\right]}{P} + \frac{2N}{P} (P-1) + \log P - 1
\]

\[
= \frac{N}{P} \left[\frac{12}{7} \log \left(\frac{N+1}{M+2}\right) - \log P + 2P - 2\right] + \frac{1}{P} \left[\frac{12}{7} \log \left(\frac{N+1}{M+2}\right) - \log P\right] + \log P - 1. \quad (7.2.1.8)
\]

The speed-up ratio of Prog '7.1 now becomes:

\[
S'_p = \frac{T_1}{T'_P}
\]

\[
= \frac{\frac{12}{7} \log \left(\frac{N+1}{M+2}\right) + \frac{12}{7} \log \left(\frac{N+1}{M+2}\right)}{\frac{12}{7} \log \left(\frac{N+1}{M+2}\right) - \log P + 2P - 2}
\]

\[
= \frac{1}{\frac{12}{7} \log \left(\frac{N+1}{M+2}\right) - \log P + 2P - 2} + \frac{\frac{12}{7} \log \left(\frac{N+1}{M+2}\right) - \log P + 2P - 2}{T'_P}
\]

\[
\left(7.2.1.9\right)
\]

The quantities in the numerators of the last two terms in equation (7.2.1.9) are very much less than \(T'_P\). Therefore, the speed-up ratio of this algorithm is of \(O\left(\frac{12}{7} \log \left(\frac{N+1}{M+2}\right) - \log P + 2P - 2\right)\).

The efficiency \(E_p\) of this algorithm is given by:

\[
E_p = \frac{S'_p}{P} = O\left(\frac{12}{7} \log \left(\frac{N+1}{M+2}\right) - \log P + 2P - 2\right). \quad (7.2.1.10)
\]

It can be easily checked that the speed-up ratio decreases when \(P\) increases which follows that the efficiency decreases as well. However, when \(P=2\) approximately a linear speed-up is obtained. But when \(P=4\), the speed-up ratio falls away from a linear speed-up. This means that the speed-up decreases qualitatively on doubling the number of processors. This is also
clear from the theoretical values of the speed-up of equation (7.2.1.9) which are listed in Table 7.2.

<table>
<thead>
<tr>
<th>N</th>
<th>M</th>
<th>P</th>
<th>Speed-up</th>
<th>EP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Theory</td>
<td>Experiment</td>
</tr>
<tr>
<td>(1024)</td>
<td>5</td>
<td>2</td>
<td>1.79</td>
<td>1.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2.34</td>
<td>2.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>2.72</td>
<td>2.37</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>2</td>
<td>1.78</td>
<td>1.57</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2.31</td>
<td>1.90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>2.68</td>
<td>2.22</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>2</td>
<td>1.76</td>
<td>1.52</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2.26</td>
<td>1.93</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>2.59</td>
<td>2.14</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>2</td>
<td>1.75</td>
<td>1.53</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2.23</td>
<td>1.89</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>2.55</td>
<td>2.12</td>
</tr>
</tbody>
</table>

**TABLE 7.2:** The theoretical values of the speed-up of Prog 7.1

The theoretical values of the speed-up of Prog 7.1 are in agreement with the experimental values. The non-linear values of the speed-up are mainly due to the logarithmic strategy that synchronises the paths (processes) at each step of the partitioning process. Also, the critical section that exists within each path to update the queue's information acts as a bottleneck that degrades the performance of the algorithm. These two terms generate a significant overhead which is measured from the run of 4 processors and listed in Table 7.3.
Table 7.3 includes the actual measurements obtained from the NEPTUNE system as well as the predicted results of the shared data and the parallel path accesses. However, the predicted results are in good agreement with those obtained from the actual 4 processors run. As discussed in the previous chapters these results were obtained using the resources measurements of the NEPTUNE system obtained from Table 5.10 (see Chapter 5). The parallel path access rate was predicted where the number of comparisons in the partitioning procedure were considered as floating point operations (flops). This number is predicted when \( P \) processors are in use. Thus, when the number of comparisons in the partitioning stage is \( (N+1) \), it becomes \( \frac{N}{P}+1 \) if \( P \) processors are co-operating. However, the corresponding loss is measured when \( P=4 \) and this equals the static loss obtained from the actual run. Similarly, the access to the shared data is predicted from the program so that \( 2\left(\frac{N}{P}+1\right) \) accesses to the shared data per \( \frac{N}{P}+1 \) operations have been made. This yields a 2:1 access rate.

Now, if we sum up the static and contention losses of the parallel path control and the critical section together with the shared data access losses, we see that this algorithm loses -3% of its efficiency.
7.3 PARALLEL DISTRIBUTIVE PARTITIONING SORTING METHODS

Some parallel versions of existing Distributive Partitioning Sorting algorithms are developed and discussed in this section. The experimental results of each version is presented together with the overheads of using a parallel system which is associated with each version.

7.3.1 The Parallel DPS: Version 1

The original DPS algorithm of Dobosiewicz [1978] divides the set of N elements to be sorted into N buckets. This strategy has the disadvantage of having a large number of buckets which contain just one element or none. Therefore, the algorithm proposed by Allison and Noga [1982] was a good base to reduce this disadvantage. However, their algorithm divides the set into N/2 buckets and uses a linear insertion sort for a bucket of smaller size.

The first parallel implementation of the DPS which we define as Version 1 follows the strategy proposed by Allison and Noga [1982]. However, three slightly different strategies of Version 1 were implemented. All the strategies are similar in their algorithmic nature as will be shown below.

The proposed algorithm of Version 1 includes three parts: a part to find the minimum (min) and maximum (max) elements of the N elements, a part to distribute the N elements into a number of buckets, and the final part is to sort these buckets by using the Quicksort (the median of three versions) if the size of the bucket is large and the linear insertion sort for a size less than M where 5 ≤ M ≤ 25. Since this algorithm is to be implemented on the NEPTUNE MIMD-system, therefore parallelism can be introduced in different parts of the algorithm. The first strategy of this algorithm which is defined as "Prog 7.2" and listed in Appendix A as
Program A.11, is now presented.

The first part is performed sequentially, i.e. by one processor only, and the min and max elements which are found in this part are kept in the shared memory to be accessed by all processors. Since it is convenient to have an equal amount of work in each path, therefore, we form smaller subsets of size \( NP = N/k \), where \( k \) is the number of the paths input to the program. Hence, in this case, we can generate \((NB=N/2)\) buckets of elements. However, in the second part of the algorithm which can be performed in parallel, \( NB \) paths are generated to distribute the elements into \( NB \) buckets, where each path works on a subset of size \((2k)\).

The distribution of an element \( a_i \) to be placed into bucket number \( j \) is performed according to the formula:

\[
j = \text{IFIX} \left( \frac{(a_i - \text{min}) \times (\text{max} - \text{min})}{0.001} + 1.0 \right).
\]

(7.3.1.1)

The quantity 0.001 is needed to ensure that the maximum element is placed into the bucket numbered \( NB \) instead of bucket (\( NB+1 \)). The last term, 1.0, in the inequality (7.3.1.1) is needed to ensure that \( j \) never becomes zero. The whole value of \( j \) is truncated to be always an integer number.

Two lists of pointers are required in this algorithm to represent the elements of each bucket. The first list \( \text{LH} \) is of size \( NB \) and is used to represent the list head for each bucket. The second list \( \text{LINK} \) which is of size \( N \) represents the links amongst the elements which belong to one bucket. Therefore, when a path calculates \( j \) for an element \( a_i \), the two lists are updated accordingly. Let \( A \) denote the input set of \( N \) elements. If the list \( \text{LH} \) is initialised to zero, then the following DO-loop that each path performs is used to distribute the elements. Thus,

\[
\text{DO } 1 \text{ I} = \text{IS}, \text{IE} \\
\quad \text{J} = \text{IFIX} ((\text{A(I)} - \text{MIN}) \times (\text{NB-0.001}) + 1.0) \\
\quad \text{LINK} (\text{I}) = \text{LH} (\text{J}) \\
\quad \text{LH} (\text{J}) = \text{I} \\
\text{1 CONTINUE}
\]
In this loop, IS and IE represent the start and end positions of the subset of the path.

The update of the LINK and LH lists should be completed as a critical section since both of the lists are kept in the shared memory and each path distributes its elements into any bucket of the NB buckets. When all the paths complete their distributions, they are synchronised. Thus, by the end of the second part we obtain NB buckets of elements where the contents of two consecutive buckets are such that all the elements of the first one are guaranteed to be smaller than those of the second one.

In the third part of the algorithm, where NB paths are generated, each path first arranges the elements of its corresponding bucket into a local vector B. This arrangement is performed by obtaining the list head of the bucket from LH and by following the corresponding links from the LINK list the corresponding elements are copied from the shared vector A into the local vector B. When all elements of the bucket are copied into B, they are sorted by using the median of three version of the Quicksort algorithm which was presented in Section (7.2) if the size of the bucket exceeds M, where \(5 \leq M \leq 25\). Otherwise, a linear insertion sort algorithm is used which is an efficient algorithm for a file of a very small size. Since, the sorting is performed in the local vector B, therefore the accesses to the shared memory are reduced. When the sorting of a bucket is complete, its elements are moved to their specific subset of locations in the shared vector C which is of size N. The vector C will then hold the final sorted elements when all the buckets have been sorted.

Experiments have been carried out on this version on the NEPTUNE system where the set of elements to be sorted is randomly generated by using a random generator in the interval \([0,1]\). These experiments were performed for \(N=1024\) elements, \(M=5\) and for a different number of paths \(k\), where \(k=N/2^m\).
where \( m \) is any integer \( \geq 1 \). The experimental results of Prob 7.2 are listed in Table 7.4. In this implementation, when \( k \) increases, the number of buckets decreases. For example, when \( k=128 \), the number of buckets \( NB=4 \).

In this case, if the number of processors \( P \) is 4, then all the \( NB \) buckets (paths) are carried out simultaneously and not one waits for a processor. On the other hand, when \( NB>P \), then only the first \( P \) paths are carried out by the \( P \) processors and \( (NB-P) \) paths are kept waiting for any available processor. This means that a significant overhead will occur which will reduce the efficiency of the algorithm. From Table 7.4, we notice that the time spent in the distribution part slightly decreases as \( k \) increases, i.e. as \( NB \) decreases. Whereas, in the sorting part, the time increases with increasing \( k \). This is because when \( k \) is small, \( NB \) is large and each bucket contains a small number of elements to be sorted. Actually, there are many paths waiting for a processor to be free which yields a certain amount of overhead, but the work performed in each path is also small in which case the processors will be available soon.

Since the min and max of the set are found by one processor, therefore, we add the time spent in the distribution and sorting parts only to obtain the total time as these two paths are performed in parallel. Thus, the relevant speed-up ratio is measured as a ratio of the total time for one processor's run and the total time obtained when \( P \) processors are in use, i.e., where \( P=2,3 \) and 4.

The Analysis of the Algorithm

In order to analyse this algorithm, the expected number of comparisons which determines the time-complexity of the algorithm is derived when the input vector is uniformly distributed.

Let \( N \) be the data size, \( k \) the number of paths and \( NB \) be the number of buckets which equals \((N/2k)\). Thus, it takes \( O(N) \) time to find the min and
<table>
<thead>
<tr>
<th>M (for linear insertion)</th>
<th>No.of Paths(k)</th>
<th>No.of Processors(P)</th>
<th>Time for finding min &amp; max</th>
<th>Time for distribution(A)</th>
<th>Time for sorting(B)</th>
<th>Total time (A+B)</th>
<th>Total Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>1.350</td>
<td>2.880</td>
<td>2.640</td>
<td>5.520</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1.460</td>
<td>1.540</td>
<td>3.00.</td>
<td>1.84</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.060</td>
<td>1.150</td>
<td>2.21.</td>
<td>2.50</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.980</td>
<td>0.900</td>
<td>1.880</td>
<td>2.94</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1.350</td>
<td>2.810</td>
<td>3.330</td>
<td>6.140</td>
<td>1.87</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.420</td>
<td>1.870</td>
<td>3.290</td>
<td>2.64.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.030</td>
<td>1.300</td>
<td>2.330</td>
<td>2.69</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.970</td>
<td>1.100</td>
<td>2.070</td>
<td>2.97</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1.350</td>
<td>2.770</td>
<td>4.010</td>
<td>6.780</td>
<td>1.89</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.400</td>
<td>2.190</td>
<td>3.590</td>
<td>2.69</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.010</td>
<td>1.510</td>
<td>2.520</td>
<td>3.18</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.990</td>
<td>1.140</td>
<td>2.130</td>
<td>3.18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>1.350</td>
<td>2.750</td>
<td>4.790</td>
<td>7.540</td>
<td>1.94</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.390</td>
<td>2.490</td>
<td>3.880</td>
<td>2.63</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.010</td>
<td>1.860</td>
<td>2.870</td>
<td>3.32</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.940</td>
<td>1.330</td>
<td>2.270</td>
<td>3.32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>1.350</td>
<td>2.710</td>
<td>5.670</td>
<td>8.380</td>
<td>1.93</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.370</td>
<td>2.970</td>
<td>4.340</td>
<td>2.64</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.030</td>
<td>2.140</td>
<td>3.170</td>
<td>3.43</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.930</td>
<td>1.510</td>
<td>2.440</td>
<td>3.43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>1</td>
<td>1.350</td>
<td>2.710</td>
<td>6.210</td>
<td>8.920</td>
<td>1.93</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.360</td>
<td>3.250</td>
<td>4.610</td>
<td>2.63</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.030</td>
<td>2.360</td>
<td>3.390</td>
<td>2.63</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.930</td>
<td>1.670</td>
<td>2.600</td>
<td>3.43</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE 7.4: The experimental results of Prog 7.2: The parallel DPS Version 1, First Strategy
max elements. However, for the sorting part of the algorithm, the time required to sort a single bucket consisting of $i$ elements will be proportional to $i \log_i i$ (see Section 7.2). Since the input is uniformly distributed, then the probability that an element belongs to a given bucket is given by $1/(NB)$, i.e. $1/(N/2k) = \frac{2k}{N}$. However, the probability that a bucket will consist of $i$ elements after $N$ elements have been distributed is obtained from the binomial distribution

$$P(i) = \binom{N}{i} \left(\frac{2k}{N}\right)^i \left(1 - \frac{2k}{N}\right)^{N-i}.$$  

(7.3.1.2)

The expected time to sort a single bucket of $N$ or more elements is given by:

$$\sum_{i=M}^{N} i \log_i i \binom{N}{i} \left(\frac{2k}{N}\right)^i \left(1 - \frac{2k}{N}\right)^{N-i}.$$  

(7.3.1.3)

Thus, the time-complexity $T_{SI}$ required to sort all the buckets in a uni-processor is given by:

$$T_{SI} = \left(\frac{N}{2k}\right) \sum_{i=M}^{N} i \log_i (\frac{N}{i}) \left(\frac{2k}{N}\right)^i \left(1 - \frac{2k}{N}\right)^{N-i}$$

$$= \left(\frac{N}{2k}\right) \sum_{i=M}^{N} \frac{i \log_i (2k)^i \cdot N(N-1) \cdots (N-i+1)}{i!} \left(1 - \frac{2k}{N}\right)^{N-i}$$

$$< \left(\frac{N}{2k}\right) \sum_{i=M}^{N} \frac{(i \log_i (2k) - 1)}{i!} = N \sum_{i=M}^{N} \frac{(i \log_i (2k) - 1)}{(i-1)!}$$

$$< N \sum_{i=1}^{\infty} \frac{(2k)^{i-1}}{(i-1)!}$$

$$< kN e^{2k}.$$  

(7.3.1.4)

The insertion sort algorithm that is used to sort the subsets of small size requires the worst case time-complexity as follows: Since $\frac{N}{2k}$ buckets are to be sorted, then if we suppose that each bucket has $\frac{2k}{(M-1)}$ subsets to be sorted by the insertion sort where $(M-1)^2$ comparisons are required to sort one subset. Therefore, in the uni-processor run:

$$T_{SI} = \left(\frac{N}{2k}\right) \cdot \frac{2k}{(M-1)} (M-1)^2 = N(M-1),$$  

(7.3.1.5)

and in the parallel run:
When the algorithm runs on a P processor system, the number of paths will be divided amongst the P processors. Thus, each processor carries out at most \( \left\lfloor \frac{N}{2kP} \right\rfloor \) paths, i.e. at most \( \left\lfloor \frac{N}{2kP} \right\rfloor \) buckets are sorted by each processor. Therefore, the expected time-complexity \( T_{SP} \) when P processors are in use is given by:

\[
T_{SP} = \left\lfloor \frac{N}{2kP} \right\rfloor \sum_{i=M}^{N} \frac{i \log e (N-i)}{N^i} \left( \frac{2k}{N} \right)^i \left( 1 - \frac{2k}{N} \right)^{N-i}.
\]

Now, we measure the time-complexity for the distribution part. It is known that distributing \( N \) elements into a number of buckets requires \( aN \) time, for some constant \( a \) (Allison and Noga [1982]). Hence, in our implementation, \( \left( \frac{N}{2k} \right) \) paths are generated in the distribution part where each path allocates a subset of size \( (2k) \), thus the complexity of each path is \( 2ak \). Therefore, the total time-complexity \( T_{d1} \) for the distribution part is given by:

\[
T_{d1} = \frac{N}{2k} \cdot 2ak = aN.
\]

When the distribution part is performed in P processors, at most \( \left\lfloor \frac{N}{2kP} \right\rfloor \) paths are carried out by each processor then the total time-complexity \( T_{dp} \) for the parallel run becomes:

\[
T_{dp} = \left\lfloor \frac{N}{2kP} \right\rfloor 2ak < a \frac{N}{P} + 1.
\]

Therefore, if we add the results of \( T_{SI} \), \( T_{d1} \) and \( T_{dp} \), we obtain the total
time-complexity $T_1$ of Prog 7.2 when run on a uni-processor. Thus,

$$T_1 < kN \cdot e^{2k} + \alpha N + N(M-1),$$

if $\alpha = 1$, then

$$T_1 < N[k \cdot e^{2k} + M].$$  \hspace{1cm} (7.3.1.10)

When we add the results of $T_{SP}$, $T_{dp}$ and $T_{IP}$, we obtain the total time-complexity for the parallel run. Thus,

$$T_P < \frac{N}{P} k \cdot e^{2k} + \frac{\alpha N}{P} + \frac{N}{P}(M-1) + 3,$$

$$T_P < \frac{N}{P} [k \cdot e^{2k} + M] + 3, \text{ if } \alpha = 1.$$  \hspace{1cm} (7.3.1.11)

The speed-up ratio, now, can be measured to give:

$$S_P = \frac{T_1}{T_P} < \frac{N[k \cdot e^{2k} + M]}{N/P[k \cdot e^{2k} + M] + 3} < P(1 - \frac{3}{T_P}) < O(P).$$  \hspace{1cm} (7.3.1.12)

From this formula, we notice that the theoretical speed-up is of $O(P)$ for all values of $k$. Whereas, from the experimental results, the speed-up is slightly less than $O(P)$ especially for large $k$. However, when $k$ is small, i.e. the number of buckets is large, the speed-up is less than $O(P)$ and this is because of the synchronisation overheads obtained when the paths carrying out the buckets are synchronised. The overhead measurements will be presented later in this chapter.

For the second strategy, of Version 1, Prog 7.3 is implemented so that the number of buckets is as given in the previous program, i.e. $NB = \frac{N}{2k}$. The difference now is that in the distribution part, $k$ paths, instead of $NB$ paths, are generated to distribute the elements into $NB$ buckets. This means that each path allocates a subset of elements of size $\left(\frac{N}{K}\right)$, whereas in Prog 7.2, the size of the subset was $2k$. The remainder of Prog 7.3 is
exactly similar to that of Prog 7.2. The experimental results of this implementation are listed in Table 7.5, where the data size is $N=1024$.

From Table 7.5, we see that the time required for the sorting part is similar to that of Prog 7.2, but the time required in the distribution part is different. However, it slightly increases when $k$ increases in this case, where the reverse was true for Prog 7.2. The speed-up in Table 7.5 is slightly greater than that of Prog 7.2 in Table 7.4. This is mainly due to the fact that a smaller number of paths is generated for the distribution part in Prog 7.3, which means, a smaller amount of overhead is achieved. Thus, the time spent in the distribution part is as follows:

$$T_{d_1} = k \cdot a \frac{N}{k} = aN \quad \text{(in the sequential run)},$$

and

$$T_{d_p} = \left[ \frac{k}{P} \right] a < \frac{N}{k} < \frac{N}{P} + 1 \quad \text{(in the parallel run)},$$

which are, obviously, similar to that of Prog 7.2. Hence, the same speed-up formula will hold which is of $O(P)$.

The third strategy of Version 1 of the parallel DPS, which we have implemented and defined as Prog 7.4, is described as follows: Let $k$ be the number of paths, then $k$ paths will be generated in the distribution part to distribute the elements into $k$ buckets. In this case, each path allocates a subset of elements of size $\binom{N}{k}$. After the distribution part is complete, $k$ paths will be generated to sort the $k$ buckets. Here, each path allocates the elements belonging to its bucket into a local vector as in the first strategy, Prog 7.2. The path, then, sorts its bucket on the local vector. Thus, at the end of this part all the buckets are sorted and the elements are in their final position in the resultant shared vector.

Thus, this strategy is different from the first strategy by the number of buckets it forms. In the first strategy, Prog 7.2, $\frac{N}{2k}$ buckets
<table>
<thead>
<tr>
<th>M (for insertion sort)</th>
<th>No.of Paths (k)</th>
<th>No.of Processors (P)</th>
<th>Time for finding min &amp; max</th>
<th>Time for distribution (A)</th>
<th>Time for sorting (B)</th>
<th>Total Time (A+B)</th>
<th>Total Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>1</td>
<td>1.350</td>
<td>2.760</td>
<td>2.630</td>
<td>5.390</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1.380</td>
<td>1.510</td>
<td>2.890</td>
<td>2.10</td>
<td>1.86</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.400</td>
<td>1.170</td>
<td>2.570</td>
<td></td>
<td>2.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.990</td>
<td>0.920</td>
<td>1.910</td>
<td></td>
<td>2.82</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1.350</td>
<td>2.770</td>
<td>3.330</td>
<td>6.100</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1.390</td>
<td>1.870</td>
<td>3.260</td>
<td></td>
<td>1.87</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.100</td>
<td>1.310</td>
<td>2.410</td>
<td></td>
<td>2.53</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.00</td>
<td>0.990</td>
<td>1.990</td>
<td></td>
<td>3.07</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1.350</td>
<td>2.790</td>
<td>4.010</td>
<td>6.800</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1.400</td>
<td>2.180</td>
<td>3.580</td>
<td></td>
<td>1.90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.100</td>
<td>1.550</td>
<td>2.660</td>
<td></td>
<td>2.56</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.940</td>
<td>1.170</td>
<td>2.110</td>
<td></td>
<td>3.22</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>1</td>
<td>1.350</td>
<td>2.840</td>
<td>4.800</td>
<td>7.640</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1.430</td>
<td>2.440</td>
<td>3.870</td>
<td></td>
<td>1.97</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.040</td>
<td>1.830</td>
<td>2.870</td>
<td></td>
<td>2.66</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.00</td>
<td>1.360</td>
<td>2.360</td>
<td></td>
<td>3.24</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>1</td>
<td>1.350</td>
<td>2.930</td>
<td>5.690</td>
<td>8.620</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1.490</td>
<td>2.980</td>
<td>4.470</td>
<td></td>
<td>1.93</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.100</td>
<td>2.160</td>
<td>3.260</td>
<td></td>
<td>2.64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.970</td>
<td>1.550</td>
<td>2.520</td>
<td></td>
<td>3.42</td>
</tr>
<tr>
<td>128</td>
<td>1</td>
<td>1</td>
<td>1.350</td>
<td>3.160</td>
<td>6.230</td>
<td>9.390</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1.590</td>
<td>3.260</td>
<td>4.850</td>
<td></td>
<td>1.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.140</td>
<td>3.100</td>
<td>4.190</td>
<td></td>
<td>2.24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.020</td>
<td>1.700</td>
<td>2.720</td>
<td></td>
<td>3.45</td>
</tr>
</tbody>
</table>

**TABLE 7.5:** The experimental results of Prog 7.3, the Second Strategy
are generated while here only k buckets are formed. The experimental results of Prog 7.4 are listed in Table 7.6.

From the implementation of this strategy, it is clear that the amount of synchronisation overheads is small compared to that of the other two strategies. Here, also the time for sorting decreases as k increases and the largest time is required when k=4, which approximately equals the time for the first strategy, when k=128. Thus, more or less the speed-up ratios are almost equal to the ratios of the other strategies. The analysis of this strategy can be carried out in a similar manner to the analysis of Prog 7.2 presented earlier.

7.3.2 The Parallel DPS with Merging Algorithm: Version 2

So far we have described three strategies of Version 1 of the DPS in which the distribution of elements into m buckets is done so that the buckets will be sorted within themselves and amongst each other. This is because the distribution is carried out according to the only one value for each of the min and max elements which are kept in the shared memory so that each path can access them. However, for a different implementation we illustrate Prog 7.5 which is listed as Program A.12 in Appendix A.

This implementation includes two main parts, one part for the distribution and the sorting, while the other part is to merge the subsets obtained from the first part. The merge part is needed here because, the input vector of N elements is divided into k subsets of size $\frac{N}{k}$ each. Within each subset, the maximum and minimum elements for that subset are calculated so that the distribution of the elements can be performed independently from the other distributions carried out by the other subsets. Also, each subset will contain k buckets after the distribution process is complete. In this case, the sorting process commences within the subset to
<table>
<thead>
<tr>
<th>M (for insertion sort)</th>
<th>No.of Paths(k)</th>
<th>No.of Processors(P)</th>
<th>Time for finding max &amp; min</th>
<th>Time for distribution(A)</th>
<th>Time for sorting(B)</th>
<th>Total time (A+B)</th>
<th>Total Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>1</td>
<td>1.350</td>
<td>2.720</td>
<td>6.240</td>
<td>8.960</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td>1.410</td>
<td>3.200</td>
<td>4.610</td>
<td>1.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td>1.060</td>
<td>3.160</td>
<td>4.220</td>
<td>2.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td>0.940</td>
<td>1.670</td>
<td>2.610</td>
<td>3.43</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1.350</td>
<td>2.710</td>
<td>5.800</td>
<td>8.510</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td>1.370</td>
<td>2.950</td>
<td>4.340</td>
<td>1.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td>1.060</td>
<td>2.240</td>
<td>3.300</td>
<td>2.58</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td>0.970</td>
<td>1.500</td>
<td>2.470</td>
<td>3.44</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1.350</td>
<td>2.750</td>
<td>5.220</td>
<td>7.970</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td>1.370</td>
<td>2.730</td>
<td>4.100</td>
<td>1.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td>1.100</td>
<td>1.810</td>
<td>2.910</td>
<td>2.74</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td>1.040</td>
<td>1.320</td>
<td>2.3600</td>
<td>3.38</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>1</td>
<td>1.350</td>
<td>2.770</td>
<td>4.730</td>
<td>7.500</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td>1.390</td>
<td>2.460</td>
<td>3.850</td>
<td>1.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td>1.040</td>
<td>1.670</td>
<td>2.710</td>
<td>2.77</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td>0.970</td>
<td>1.240</td>
<td>2.210</td>
<td>3.39</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>1</td>
<td>1.350</td>
<td>2.800</td>
<td>5.510</td>
<td>8.310</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td>1.430</td>
<td>2.810</td>
<td>4.240</td>
<td>1.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td>1.040</td>
<td>1.830</td>
<td>2.870</td>
<td>2.89</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td>0.990</td>
<td>1.360</td>
<td>2.350</td>
<td>3.54</td>
</tr>
</tbody>
</table>

**TABLE 7.6: The experimental results of Prog 7.4: the Third Strategy**
sort all these buckets in a similar manner as described before. One point to mention here is that the two lists LH and LINK, which are used to store the pointers for the elements being distributed into the buckets, are held in the local memory rather than the shared memory, since each subset has its own buckets to sort. Therefore, with this method some space in the shared memory is saved but an output vector which is held in the shared memory is still needed.

The merge procedure commences when all the subsets have been sorted. However, the two-way merge algorithm (Section 6.3.1) is used in this version. The experimental results of the implementation are listed in Table 7.7.

From Table 7.7, we notice that the total time which includes the sorting and merging times is greater than that of Prog 7.4 of Version 1. However, one should notice that the minimum and maximum elements are found within the sorting part. This is why the total time becomes greater than the total time of Prog 7.4 which did not include the time for finding those two elements. On the other hand, the speed-up of this method is less than the speed-up of Prog 7.4. Here, the two-way merge algorithm whose performance description was presented in detail earlier in Section (6.3.1), degrades the performance and reduces the amount of parallelism one can achieve.

Now, in order to analyse this algorithm, we proceed as follows: in the sorting part of the algorithm, the distribution and sorting of the elements are carried out by k paths where only k buckets are formed in each path. Each path sorts its buckets which yield a sorted subset of size \( \frac{N}{k} \). In this analysis, we do not assume that the buckets have a random number of elements according to a specific probability function, since we know that the total number of elements is \( \frac{N}{k} \) in the path. Instead, we derive the complexity of the whole path where the Quicksort algorithm is used to sort the buckets of that path.
<table>
<thead>
<tr>
<th>M (for insertion sort)</th>
<th>No. of Paths (k)</th>
<th>No. of Processors (P)</th>
<th>Time for sorting</th>
<th>Time for merging</th>
<th>Total time</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>4</td>
<td>7.690</td>
<td>1.660</td>
<td>9.350</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>3.890</td>
<td>1.270</td>
<td>5.160</td>
<td>1.81</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3.820</td>
<td>1.270</td>
<td>5.090</td>
<td>1.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>2.00</td>
<td>1.270</td>
<td>3.270</td>
<td>2.86</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>6.110</td>
<td>2.490</td>
<td>8.600</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>3.100</td>
<td>1.690</td>
<td>4.790</td>
<td>1.795</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2.350</td>
<td>1.690</td>
<td>4.040</td>
<td>2.13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.800</td>
<td>1.490</td>
<td>3.290</td>
<td>2.61</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1</td>
<td>4.640</td>
<td>3.270</td>
<td>7.910</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2.370</td>
<td>2.110</td>
<td>4.480</td>
<td>1.77</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.780</td>
<td>1.990</td>
<td>3.770</td>
<td>2.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.740</td>
<td>1.340</td>
<td>3.080</td>
<td>2.57</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>4.180</td>
<td>4.050</td>
<td>8.230</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2.120</td>
<td>2.490</td>
<td>4.610</td>
<td>1.78</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.470</td>
<td>2.260</td>
<td>3.730</td>
<td>2.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.130</td>
<td>2.020</td>
<td>3.150</td>
<td>2.61</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1</td>
<td>4.640</td>
<td>4.840</td>
<td>9.480</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2.340</td>
<td>2.890</td>
<td>5.230</td>
<td>1.81</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.620</td>
<td>2.550</td>
<td>4.170</td>
<td>2.27</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.200</td>
<td>2.130</td>
<td>3.330</td>
<td>2.85</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>1</td>
<td>6.080</td>
<td>5.630</td>
<td>11.710</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>3.080</td>
<td>3.300</td>
<td>6.380</td>
<td>1.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2.070</td>
<td>2.810</td>
<td>4.880</td>
<td>2.40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.610</td>
<td>2.610</td>
<td>4.220</td>
<td>2.77</td>
</tr>
</tbody>
</table>

**TABLE 7.7:** The experimental results of Version 2: the Parallel DPS with Merging algorithm
Since we have the time-complexity of the Quicksort as obtained from formula (7.2.1.3), therefore, to sort \( \frac{N}{k} \) elements by Quicksort requires the following complexity when only one processor is used. Thus,

\[
T_{S1} = k \left( \frac{N}{k} + 1 \right) \left( \frac{12}{7} \log_e \left( \frac{N/k + 1}{M+2} \right) \right) < N \left( \frac{12}{7} \log_e \left( \frac{N/k + 1}{M+2} \right) \right) + \frac{12}{7} k \log_e \left( \frac{N/k + 1}{M+2} \right).
\]

Since the distribution and the locating of the minimum and maximum of the \( \frac{N}{k} \) elements requires \( a \frac{N}{k} \) time, where \( a \) is a constant, then \( aN \) time is required by the \( k \) paths. If we add this quantity to \( T_{S1} \), we then obtain the total time-complexity of the single processor run. Thus, \( T_{S1} \) becomes:

\[
T_{S1} < N \left( \frac{12}{7} \log_e \left( \frac{N/k + 1}{M+2} \right) + 1 \right) + \frac{12}{7} k \log_e \left( \frac{N/k + 1}{M+2} \right), \quad (7.3.2.1)
\]

where \( a \) is assumed to be 1.

On the other hand, in the parallel run, each processor carries out at most \( \left\lfloor \frac{k}{p} \right\rfloor \) paths where the same Quicksort complexity mentioned above is required in each path. If we consider the time required for the distribution which is now \( a \frac{N}{p} \), then the total complexity \( T_{SP} \) for the sorting part is given by:

\[
T_{SP} < \left\lfloor \frac{k}{p} \right\rfloor \left( \frac{N}{k} + 1 \right) \left( \frac{12}{7} \log_e \left( \frac{N/k + 1}{M+2} \right) \right) + \frac{N}{p} < \frac{N}{p} \left( \frac{12}{7} \log_e \left( \frac{N/k + 1}{M+2} \right) + 1 \right) + \frac{12}{7} k \log_e \left( \frac{N/k + 1}{M+2} \right) + 1 \quad (7.3.2.2)
\]

For the merge procedure, the time-complexity for the sequential and parallel runs are obtained from the pre-derived formulae (6.3.1.1) and (6.3.1.4). Therefore, the total time-complexity \( T_1 \) of the algorithm in the sequential run becomes:

\[
T_1 < N \left( \frac{12}{7} \log_e \left( \frac{N/k + 1}{M+2} \right) + 1 \right) + \frac{12}{7} \log_e \left( \frac{N/k + 1}{M+2} \right) + N \log k - k+1 < N \left( \frac{12}{7} \log_e \left( \frac{N/k + 1}{M+2} \right) + \log k + 1 \right) + \frac{12}{7} \log_e \left( \frac{N/k + 1}{M+2} \right) - k+1, \quad (7.3.2.3)
\]

and the total time-complexity \( T_P \) of the parallel run becomes:
\[ T_p < \frac{N}{P} \left[ \frac{12}{7} \log_e \left( \frac{N}{k+1} \right) + 1 \right] + \frac{12}{7} \log_e \left( \frac{N}{k+1} \right) + 1 + \frac{N}{P} \log \left( \frac{k}{P} \right) + \]
\[ \frac{2N}{P} (P-1) - \frac{k}{P} + 1 + \log \left( \frac{k}{P} \right) \]
\[ < \frac{N}{P} \left[ \frac{12}{7} \log_e \left( \frac{N}{k+1} \right) + \log \left( \frac{k}{P} \right) + 2P - 1 \right] + \frac{12}{7} \log_e \left( \frac{N}{k+1} \right) - \frac{k}{P} \]
\[ + 1 + \log \left( \frac{k}{P} \right) \right]. \quad (7.3.2.4) \]

Then, the speed-up ratio is given by:

\[ S_p = \frac{T_1}{T_p} \]
\[ < \left( \frac{N \left[ \frac{12}{7} \log_e \left( \frac{N}{k+1} \right) + \log \left( \frac{k}{P} \right) + 2P - 1 \right] + \frac{12}{7} \log_e \left( \frac{N}{k+1} \right) - \frac{k}{P} + 1 + \log \left( \frac{k}{P} \right) }{ \frac{P \left[ \frac{12}{7} \log_e \left( \frac{N}{k+1} \right) + \log \left( \frac{k}{P} \right) + 2P - 1 \right] + \frac{12}{7} \log_e \left( \frac{N}{k+1} \right) - \frac{k}{P} + 1 + \log \left( \frac{k}{P} \right) } \right). \quad (7.3.2.5) \]

The theoretical values of the speed-up are listed in Table 7.8.

<table>
<thead>
<tr>
<th>N</th>
<th>M</th>
<th>No.of Processors(P)</th>
<th>No.of Paths (k)</th>
<th>Total Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Theory</td>
</tr>
<tr>
<td>(1024)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>1.80</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>1.80</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>1.796</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td>1.792</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>64</td>
<td>1.789</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>2.79</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>2.77</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>2.75</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td>2.74</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>64</td>
<td>2.70</td>
</tr>
</tbody>
</table>

**TABLE 7.8:** The theoretical values of the total speed-up of Prog 7.5.

From Table 7.8, we notice that the theoretical values of the speed-up measured from the formula (7.3.2.5) are in good agreement with the
experimental results. Also, one can notice that the speed-up of this algorithm is of order less than $P$, whereas, the speed-up of the algorithm of the first strategy of Version 1 was of $O(P)$. Therefore, with the saving of the storage, which is in this case, the lists of pointers that are kept in the local rather than shared memory, there is a decrease in the speed-up which comes actually from the behaviour of the 2-way merge algorithm. Generally this algorithm is slower than the algorithms in Version 1 where only the distribution and sorting procedures were implemented. However, in order to see the difference of their actual behaviour on the NEPTUNE system, the overhead measurements were obtained to explain the difference.

7.3.3 The Performance and Overhead Measurements of the Algorithms

The parallel performance of the algorithms presented in the previous sub-sections obviously generate some overheads from the synchronisations and the shared data access. Thus, the measurements of the overheads of the actual four processors run and the predicted quantities that are measured from the program itself are listed in Table 7.9, where $N=1024$. The timing measurements of the floating point operations and the path or shared data accesses are obtained from Table (5.10) (Chapter 5).

The static parallel path access overhead is measured from the 4 processors run and also the contention. These results are in agreement with the results of the predicted formula obtained from the program. Since each algorithm of the four strategies includes two parts (for sorting and distribution or merging), therefore the overheads of each part are measured alone. Thus, the measurements are a percentage of the time of the part under consideration and not the total time of the algorithm.

Similarly, the shared data access overhead is measured for each part.
In all the algorithms, the same access to the shared data is predicted. This means that these algorithms differ only in their access to the parallel paths. It is obvious from Table 7.9 that the first strategy (Prog 7.2) yields the higher cost of the parallel path access overheads. This is because \( \frac{N}{2k} \) paths are generated in the two parts of the algorithm. Whilst Prog 7.3 requires less overheads in the distribution part since only \( k \) paths instead of \( \frac{N}{2k} \) are generated in this part. The sorting part is similar to that of Prog 7.2. In both cases, when \( k \) decreases, i.e. \( \frac{N}{2k} \) increases, the overhead increases, but for Part 1 of Prog 7.3, the overhead decreases as \( k \) decreases.

In the third strategy, the two parts of Prog 7.4 involve less overheads than the other two algorithms and this is because only \( k \) paths are generated in the two parts. In this case, when \( k \) increases the overhead increases as well. Actually, the overhead results here are in reverse to that of Prog 7.2 for an obvious reason, i.e., the value when \( k=4 \) in Prog 7.2 equals the value when \( k=128 \) in Prog 7.4.

On the other hand, in Prog 7.5 where the merge procedure is applied the overheads in both parts are less than that of the other three algorithms. The sorting part here has less overheads than, for example, Prog 7.4, since the number of operations performed in it is greater than that of Prog 7.4. These operations include finding the minimum and maximum elements and distributing the elements into buckets. Therefore, when the path has a large complexity, the path access overhead will be small.

The predicted formulae of the complexity for the path access shown in Table 7.9 are simply obtained from the programs as described previously. The term \( \frac{3N}{2k} \) of Prog 7.5 refers to the complexity of distributing the \( \frac{N}{k} \) elements which is assumed on average to be \( \frac{3N}{2k} \). This is added to the complexity of the sorting which is \( \frac{N}{k} \log \left( \frac{N}{k} \right) \) which then yields the total complexity per path.
<table>
<thead>
<tr>
<th>Program</th>
<th>Paths (k)</th>
<th>Parallel Path Control</th>
<th>Critical Section</th>
<th>Shared Data Access</th>
<th>Parallel Path Access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Static Contention</td>
<td>Static Contention</td>
<td>Rate Loss</td>
<td>Rate Loss</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Prog 7.2</td>
<td>4</td>
<td>3%</td>
<td>3%</td>
<td>0.8%</td>
<td>0.13%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.3%</td>
<td>1.4%</td>
<td>0.1%</td>
<td>0.12%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.7%</td>
<td>0.6%</td>
<td>0.08%</td>
<td>0.2%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>0.4%</td>
<td>0.3%</td>
<td>0.13%</td>
<td>0.12%</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>0.25%</td>
<td>0.16%</td>
<td>0.13%</td>
<td>0.08%</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>0.16%</td>
<td>0.09%</td>
<td>0.12%</td>
<td>0.11%</td>
</tr>
<tr>
<td>2. Prog 7.3</td>
<td>4</td>
<td>0.16%</td>
<td>3%</td>
<td>0.08%</td>
<td>0.1%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.25%</td>
<td>1.4%</td>
<td>0.13%</td>
<td>0.08%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.4%</td>
<td>0.6%</td>
<td>0.13%</td>
<td>0.3%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>0.8%</td>
<td>0.3%</td>
<td>0.13%</td>
<td>0.12%</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1.4%</td>
<td>0.15%</td>
<td>0.12%</td>
<td>0.08%</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>3%</td>
<td>0.09%</td>
<td>0.08%</td>
<td>0.24%</td>
</tr>
</tbody>
</table>

TABLE 7.9: The performance measurements of the parallel DPS algorithms on the NEPTUNE system

*CS means critical section timing measurement which on NEPTUNE is 400 μs
<table>
<thead>
<tr>
<th>Program</th>
<th>Path (k)</th>
<th>Parallel Path Control</th>
<th>Critical Section</th>
<th>Shared Data Access</th>
<th>Parallel Path Access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Static</td>
<td>Contention</td>
<td>Static</td>
<td>Contention</td>
</tr>
<tr>
<td>3. Prog 7.4</td>
<td>4</td>
<td>0.17%</td>
<td>0.09%</td>
<td>0.17%</td>
<td>0.38%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.25%</td>
<td>0.16%</td>
<td>0.13%</td>
<td>0.1%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.41%</td>
<td>0.3%</td>
<td>0.12%</td>
<td>0.24%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>0.74%</td>
<td>0.7%</td>
<td>0.03%</td>
<td>0.24%</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1.4%</td>
<td>1.4%</td>
<td>0.08%</td>
<td>0.16%</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>3%</td>
<td>3%</td>
<td>0.12%</td>
<td>0.09%</td>
</tr>
<tr>
<td>4. Prog 7.5</td>
<td>4</td>
<td>0.12%</td>
<td>0.06%</td>
<td>0.14%</td>
<td>2.5%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.18%</td>
<td>0.09%</td>
<td>0.4%</td>
<td>2.2%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.3%</td>
<td>0.19%</td>
<td>0.13%</td>
<td>2%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>0.7%</td>
<td>0.3%</td>
<td>0.15%</td>
<td>1.7%</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1.2%</td>
<td>0.6%</td>
<td>0.13%</td>
<td>1.6%</td>
</tr>
</tbody>
</table>

**TABLE 7.9 (continued)**
7.4 CONCLUSIONS

In the previous section, two versions of the parallel DPS algorithm, were proposed, where in the first version, three slightly different strategies were implemented. All the implementations required less running-time than the Quicksort algorithm (the median of three elements version) which was described and analysed in Section (7.2). The complexity of the former case is of $O(N)$, while that of the Quicksort is of $O(N \log_e N)$.

However, the parallelism is exploited more in the parallel DPS algorithms than in the parallel Quicksort. In Quicksort, when $P$ processors are used, only one processor is activated in the first partitioning step, while $(P-1)$ processors remain idle. After the $(\log P)^{th}$ step of the partitioning algorithm, all the $P$ processors are activated hence full parallelism is obtained in the proceeding steps. Meanwhile, in the parallel DPS algorithms, all the $P$ processors are used from the commencement of the algorithm and since the number of paths $k>P$, therefore, the $P$ processors are always kept busy doing useful work. This is why a higher speed-up is obtained in the parallel DPS especially for Version 1 algorithms. The speed-up of the algorithm in Version 2 is slightly higher than the speed-up of the Quicksort and this was because of the 2-way merge algorithm implemented in Version 2.

In order to see the difference in the timing results of the DPS and Quicksort algorithms, we list these results in Table 7.10. In this table, for the algorithms of Version 1, we chose $k$ so that the parallel paths generated was 4. For example, in strategy 1, when $k=128$, only 4 paths were generated. Thus, we chose the timing results when $k=128$. The numbers between the brackets represent the relative speed-ups for that algorithm.

As can be easily seen, the parallel DPS algorithms are faster than the Quicksort algorithm by:

- $14\% - 18\%$, for $P=1$
- $18\% - 32\%$, for $P=2$
- $29\% - 43\%$, for $P=4$. 
Although, it is expected that these ratios may be greater for a larger data size.

As indicated earlier, finding the maximum and minimum elements is performed sequentially, therefore this procedure can be improved if it is done in parallel then its timing result is added to the other two parts of the algorithm. This has not been implemented yet. A proposed way of doing it is to divide the original set into equal subsets. Then each processor, which carries out a subset, finds its own min and max elements. When all the subsets have their own min and max, a single processor then finds the minimum of all the minimums and the maximum of all the maximums and keeps them in the shared memory. The time for this procedure can then be added to the timing of the other two parts, hence the above ratios of the difference of the Quicksort and the DPS algorithms will then be smaller.

<table>
<thead>
<tr>
<th>P</th>
<th>Parallel Quicksort</th>
<th>Parallel DPS (Version 1)</th>
<th>Parallel DPS (Version 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6.810(1.60)</td>
<td>4.610(1.93)</td>
<td>4.850(1.94)</td>
</tr>
<tr>
<td>3</td>
<td>5.260(2.07)</td>
<td>3.390(2.63)</td>
<td>4.190(2.24)</td>
</tr>
<tr>
<td>4</td>
<td>4.590(2.37)</td>
<td>2.600(3.43)</td>
<td>2.70(3.45)</td>
</tr>
</tbody>
</table>

*TABLE 7.10: The timing results for the Quicksort and the Parallel DPS algorithms*

As the DPS algorithms have been shown to be faster than the Quicksort algorithm, they require more computer space than the Quicksort. We showed that for the strategy 1, two lists of pointers of size N and \( \frac{N}{2k} \) were required for the distribution of the elements, where N and k are, respectively, the data size and number of paths. Another list of size N was also required to hold the final sorted elements. Whilst, in the Quicksort, only the input
array of size $A$ is required besides the two queues that are used to keep the start and end pointers of each subset obtained from the partitioning procedure.

On the other hand, the overheads arising from the parallel path control in some of the DPS algorithms are greater than that of the Quicksort. However, the Quicksort has generated 0.5% overheads (see Table 7.3), whereas, the DPS algorithms have a maximum cost of -5%. This is, of course, due to the number of parallel paths generated in the latter. On the other hand, the algorithm of Version 2 of the DPS had a less overhead cost than the Quicksort. Thus, by referring to Table 7.9, when $k=4$, the losses of the two parts of the algorithm then give a sum of -0.2% which is less than the Quicksort's static loss. This last algorithm implements the DPS with the 2-way merge algorithm, where the latter generates more overheads as $k$ increases, as we described in Chapter 6.

On conclusion, one can say that, the algorithms of Version 1 of the parallel DPS demonstrated better timing results and higher speed-up ratios than that obtained from Version 2 of the parallel DPS, although, the two versions proved to be faster and more efficient than the Quicksort algorithm. The reason for this is because, the DPS has a time complexity of $O(N)$ whereas, the complexity of the Quicksort is of $O(N \log N)$. 
Chapter 8

CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK
In this thesis, we have presented a description of the architecture of the current parallel computers such as SIMD and MIMD computers and how the research in this area is developing as technology advances where the whole computer can be assembled on a silicon VLSI chip, which may contain millions of transistors. In fact, parallel processing on a single chip in either MIMD, SIMD, or pipelining fashion was recommended, where the design of such chips basically relies on the regularity of patterns for communication between active elements.

In Chapter 2, we have showed how to program parallel systems where parallelism in a program can be implicit or explicit, where in such parallelism, some parallel constructs have been added to specific languages to be used on a parallel computer. Some parallel computational models (e.g. Petri nets) have been also described to explain how such models have been used as a method of representing process synchronisation and mutual exclusion.

In general, programming parallel systems is more difficult than programming uniprocessor systems and this has led to the parallelism being concealed on most existing MIMD computers. However, recent research is being considered where the advantages of the greater speed to be obtained from the explicit use of parallelism is combined with the greater throughput and reliability that could be obtained with a system comprising several autonomous processors and concealed parallelism. This could in fact, lead to a promise that MIMD-type computers are relatively flexible. Since these computers consist of P complete computers, P independent computations can be supported simultaneously. On the other hand, the SIMD-type computers are less flexible and more specialised because all programs for these machines must be formed into a sequence of vector instructions. However, many numerical problems are easily adapted to these types of
computers, but there is still a large class of other problems that do not appear to be well-suited. Thus, the central issues behind the development of a SIMD computer deal with the efficient execution of specific problems.

Usually, communication geometry and data movement present a major role of an algorithm in SIMD computers. Meanwhile, for the VLSI implementation, it is required that the communication geometry be simple and manageable, and that silicon area rather than the number of gates only should be considered.

In fact, for MIMD computers, the techniques for programming such computers for efficient parallel operations are much less developed than the corresponding techniques for SIMD vector processors. This, however, does not mean that the class of problems that are suitable for MIMD computers can be easily implemented. The problem arising here lies in making the P computers co-operate so that one problem can be partitioned among the P processors of a multiprocessor and can be solved with greater speed than it could be done on a single processor. In order to make the multiprocessor effective, we need the speed increase to be substantial, hopefully of order P. This can be achieved when special attention is paid to the problem of minimising the synchronisations of the processors, sharing data among the processors, and scheduling the computations.

The main study in this thesis was to design and analyse parallel algorithms which are suitable for MIMD computer implementations. The experiments of our study were carried out on the NEPTUNE MIMD system at Loughborough University.

Analysing an algorithm is important for different reasons. The most straightforward reason is to discover its vital statistics in order to evaluate its suitability for various applications or compare it with
other algorithms. Generally, the vital statistics of interest are the time and space, most often time. Actually, in all the algorithms presented in this thesis, we were interested in determining how long an implementation of a particular algorithm would run on the NEPTUNE system and with some of the algorithms we determined how much space they required.

The analysis was basically done for MIMD-type computers. We concentrated on deriving results for essential characteristics of the algorithm which can be used to estimate precisely the true resource requirements on actual machines. From that analysis, we were able to measure the relative speed-up ratio of each algorithm which was the main criteria to exploit the efficiency of using parallelism.

In practice, some algorithms which use a large amount of memory space may be performed with less time than algorithms which require very little memory space as this was indicated by the parallel Quicksort and parallel DFS algorithms presented in Chapter 7.

The analysis of an algorithm can help one to understand it better, and can suggest informed improvements. However, the more complicated the algorithm, the more difficult the analysis is. More important, the careful search required for proper analysis often leads to more efficient and more correct implementations of algorithms. In fact, when the results of the analytic study agree with that of the empirical, one can then be convinced of the validity of the algorithm as well as the correctness of the process of analysis. This was what we have achieved in our study of different types of algorithms.

In Chapters 4 and 5, the asynchronous iterative methods (AJ, AGS) which included an evaluation section and a critical section to update the components, were presented. The Purely Asynchronous (PA) and Purely Asynchronous Over-Relaxation (PAOR) which have no critical sections were
proved to be faster than the other iterative methods with a linear speed-up achievement. All these iterative methods were implemented to solve the one-dimensional and two-dimensional Dirichlet problems.

Two ways of allocating subsets to processes were implemented in these algorithms: the sequential and non-sequential decomposition. The experimental results showed that the sequential decomposition was better than the non-sequential. Also, the number of iterations which is required to satisfy convergency remained unchanged when the number of processes (k) increased in the PA and PAOR methods. Whereas, in the case of AJ, AGS, ASOR methods, it increased linearly with the increase of k.

The behaviour of the relaxation factor \( \omega \) in the algorithms with critical sections was unstable. It was shown in Chapter 4 that this behaviour was related to the instability of the spectral radius of the matrix of the problem to be solved when the number of processes increased. This was due to the existence of critical sections. The PAOR method had the same \( \omega \) for any number of processes.

A new parallel block iterative method (the 4-point block method) was developed in Chapter 5. This method was implemented with three different strategies. The strategy when all blocks of 4-points were stored in a shared list and evaluated one after another, was shown to achieve a better speed-up than the other two strategies. The parallel 4-point block iteration method was empirically shown to be \( (2)^{1/2} \) faster than the PAOR method. The three strategies were implemented synchronously and asynchronously. The overhead of synchronisation, shared data access and parallel path access were measured for all three strategies. The synchronous version of the strategy of taking up 2 lines at a time required more idle time than the others (Table 5.12). This was because the processes were synchronised at the end of each iteration step, hence the performance of the algorithm was degraded.
Since the execution times for these asynchronous iterative methods were random rather than constant, then the analytic approach was via Queueing Theory and Order Statistics. This was because of the critical sections that can be accessed only by one process at a time while the others are kept waiting in the queue. Therefore, the concept of the 'birth-death' process (Kleinrock [1975]) was required to serve the processes of the algorithm. The analytic results were in very good agreement with the empirical results.

MIMD computers can also exploit parallelism for non-numeric applications usefully. We chose for our study the problems of sorting and merging of a number of subsets since they are important in many computer applications such as information processing. The methods implemented in Chapters 6 and 7 had speed-up ratios of $O\left(\frac{P \log M}{\log (\frac{M}{P}) + 2P - 2}\right)$ such as the 2-way merge and $O(P)$ such as the odd-even merge, where $P$ and $M$ are respectively, the number of processors and the number of generated paths.

In those algorithms, many paths such as $M^2P$ were generated in order to see the effectiveness of the system as well as the overheads obtained when such algorithms were implemented. From this, one can also predict what is the maximum number of paths to be generated, i.e. to what degree the problem to be solved can be partitioned among the processors, and what is the recommended number of processors that can be employed in the solution.

However, in the 2-way merge, the potentiality of parallelism decreased in each step of the algorithm, whereas almost full parallelism was achieved in each step of the odd-even merge algorithm. Therefore, the processors suffer from contention in the 2-way merge as presented in Table 6.13. Although, the odd-even merge required a larger running-time, it achieved almost a linear speed-up. On the other hand, the new merge algorithm had
a slightly larger speed-up but still the 2-way merge has the least running-time. Since we are mainly concerned with the speed-up ratio in using a parallel system, we conclude that the odd-even merge algorithm is the best amongst the others.

The odd-even merge algorithm presented larger amounts of static parallel path losses than the other merge algorithms since it required \( M \) steps to complete the merge. The shared data loss which was predicted was 0.2% for all algorithms.

The binary merge and the merge by jump searching algorithms were implemented and we noticed that merging by jump searching required less time than the binary merge though the same speed-up ratios were achieved. These algorithms were also associated with the same amount of static overheads of parallel path control.

As different versions of the merge by jump searching were applied, the version of jump sizes \((N/3)\) and \((N/9)\) was shown to be the fastest one, where \( N \) is the data size. Although no theory was found for this case and we just assumed that those jump sizes existed. Thus, we leave this as an open problem to find the theory that proves this version is faster than other jump algorithms of jump sizes \((N^\alpha)\), where \( \alpha \) is a constant. These merge algorithms were analysed by means of the probabilistic approach using some probability distributions. The results of this analysis closely matched the results obtained from the experiments.

The subsets that were to be merged by any of the above merging methods were sorted by the bubble sort or the neighbour sort algorithms. The former presented significant high speed when \( M \) paths \( \gg P \) were generated. The set to be sorted was partitioned equally amongst the paths so that the subsets were sorted within themselves and not amongst each other. Therefore, they were merged to obtain the final sorted set.
It might be interesting to know whether a set of elements can be sorted using the bubble sort alone without requiring the merge procedure. Such an idea has not been implemented yet. Although the bubble sort is of $O(N^2)$, parallel implementation might present a significant speed-up as we had in the above-mentioned implementation which produced partially sorted subsets.

For a very efficient sorting method, we implemented the Quicksort and derived two parallel versions of the Distributive Partitioning Sorting (DPS) algorithm. The latter is the fastest known sequential sorting algorithm as it is of $O(N)$. However, the empirical results obtained from the NEPTUNE system declared that the parallel DPS was faster than the parallel Quicksort although its space requirement is substantial. The analysis approach here also required some probability distributions.

Now, we would suggest that if the interval $[0,1]$ is partitioned into $k$ sub-intervals which are assigned into $k$ processes, then each process will sort its own sub-interval using the DPS algorithm. At the end, the sub-intervals and hence the whole set will be sorted. This might give better timing results.

For all the algorithms presented in this thesis, the analytic results for the time-complexity, hence the speed-up ratio were in good agreement with the empirical results. Also, the actual overhead measurements of such algorithms when implemented on the NEPTUNE system were almost the same as the results of the predicted formulas. By those measurements, we have shown that it is important to consider all the resource demands of a parallel program. Obviously, the ability of a given parallel computer to meet the resource demands of a program limits only the effectiveness of the program on that computer. However, if all computer systems are roughly similar those measurements could provide a basic
framework to the algorithm designer to realise which direction one can go in developing more suitable parallel algorithms. On the other hand, if all algorithms make approximately similar demands to some resources then the measurements guide the hardware designer to provide a system that can meet the resource demands of most algorithms. Unfortunately, this has not been achieved yet in practice.
C THIS PROGRAM IMPLEMENTS THE SYNCHRONOUS POINT JACOBI ITERATIVE METHOD.
C THE LINES OF THE MESH TO BE SOLVED ARE ASSIGNED TO NPATH SUBSETS
C SO THAT THE LINES ARE GROUPED IN A SEQUENTIAL FORM, I.E. EACH
C CONSEQUENT NUMBER OF LINES ARE IN ONE SUBSET. EACH PROCESSOR THEN
C EVALUATES ITS OWN SUBSET FOR ONE ITERATION. THEN ALL PROCESSORS ARE
C SYNCHRONISED AFTER EACH ITERATION STEP.
C THIS ALGORITHM REQUIRES ONE SHARED ARRAY (XI) AND TWO LOCAL ARRAYS
C AND AX TO HOLD THE VALUES JUST EVALUATED. A FLAG IS NEEDED IN EACH
C PROCESSOR TO PERFORM A GLOBAL CONVERGENCE TEST.

PROGRAM A.1

INTEGER*2 ITIME
DIMENSION X(26,26),HX(26,26),AX(26,26),ERROR(700),IFLAG(20),
ITIME(100)
C DEFINE THE SHARED DATA.
C READ THE MESH SIZE N, THE NUMBER OF PATHS TO BE GENERATED NPATH, AND THE
C REQUIRED DEGREE OF ACCURACY FOR THE COMPONENTS TO ACHIEVE (EPS), AND THE
C THE NUMBER OF PROCESSORS TO BE USED IN THE EVALUATION.
C INITIALISE THE BOUNDARY CONDITIONS AND THE COMPONENTS OF
C THE MESH.
DO 73 I=1,N
X(I,1)=100.0
DO 71 J=2,N
X(I,J)=0.0
CONTINUE
C ADJUST THE ITERATION COUNTER ITER AND START ITERATING UNTIL ALL OF
C COMPONENTS ARE CONVERGED, OTHERWISE UNTIL IT REACHES THE MAXIMUM NUMBER OF
C ITERATIONS GIVEN IN THE PROGRAM AS MAXITR.

CALL READST(HX)

CALL RELEAS(IPS,IPF,AX,HX)

C A CRITICAL SECTION TO UPDATE THE CORRESPONDING LOCATIONS OF
C COMPONENTS IN THE SHARED ARRAY X SO THAT OTHER PROCESSORS CAN USE THEM.

C A CRITICAL SECTION TO READ ALL THE COMPONENTS OF THE MESH AND PUT THEM
C IN THE LOCAL ARRAY HX.
CALL READST(HX)

C THIS LOOP IS EXECUTED IN PARALLEL
C CALCULATE THE COMPONENTS OF THE SUBSET USING THE JACOBI ITERATIVE METHOD
C AND ASSIGN THE RESULT INTO A LOCAL ARRAY AX.
C CHECK WHETHER THIS SUBSET AND THE OTHER SUBSETS OF OTHER PATHS ARE
C CONVERGED. IF THE CURRENT SUBSET IS CONVERGED SET ITS CORRESPONDING FLAG
C INTO 2.
DO 28 I=JL, IMM
IF(1.E.0.E. EPS) GOTO 10
IF(Iflag(IP)=2
CONTINUE
83
Iflag(IP)-2
GOTO 10

CHI!CK FOR CONVERGENCY
IN ORDER TO ITERATE AGAIN IF REQUIRED.
IF(IFLAG(IP).EQ.I)
GOTO 100
.
JL=JL+1
IF(JL.LE.NPATH) GOTO 77

END TIMING AND WRITE THE OUTPUT ARRAY X.
SPOOL 3
CALL TIMOUT(ITIME)
WRITE(6.87) ITIME
FORMAT(/2X.~TIMINa'/8(I6.2X)
WRITE(6.88) ITER
FORMAT('THE CONVERGENCE IS ACHIEVED AFTER'.IX.I4,IX.'ITERATIONS')
WRITE(6.89)
FORMATC/2X.'THE MESH IS')
DO 91 I=1,N
WRITE(6,44) (X(I,,J),J=1,N)
CONTINUE
GOTO 200
SPOOL 4
CALL TIMOUT(ITIME)
WRITE(6.87) ITIME
WRITE(6.88) ITER

A CRITICAL SECTION TO READ ALL THE COMPONENTS OF THE MESH.
SUBROUTINE READST(HY)
DIMENSION X(26,26), HY(26,26)
$SHARED X,N
NAME THE REGION TO BE USED WHICH IS PROTECTED BY $ENTER/$EXIT.
$REGION READST
$ENTER READST-
DO 23 I=1,N
DO 35 J=1,N
HY(I,J)=X(I,J)
CONTINUE
CONTINUE
DO 21 K=1,N
HJ(K1,K2)=X(K1,K2)
CONTINUE
$EXIT READST
RETURN

A CRITICAL SECTION TO UPDATE THE COMPONENTS AND RELEASE THEM TO OTHER
PROCESSORS, THEN TO READ ALL THE COMPONENTS FOR THE NEXT ITERATION.
SUBROUTINE RELEASE(I1,I2,AY,HJ)
DIMENSION X(26,26), AY(26,26), HJ(26,26)
$SHARED X,N,N1,N2
$REGION RELEASE
$ENTER RELEASE
DO 20 I=1,12
DO 34 J=1,N
X(I,J)=AY(I,J)
CONTINUE
CONTINUE
X(I1,I2)=HJ(I1,I2)
DO 22 K=1,N
HJ(K1,K2)=X(K1,K2)
CONTINUE
$EXIT RELEASE
RETURN
END
PROGRAM A.2

C THIS PROGRAM IMPLEMENTS THE ASYNCHRONOUS POINT JACOBI ITERATIVE METHOD
C TO SOLVE THE 2-DIMENSIONAL DIRICHLET PROBLEM. THE PROBLEM IS DIVIDED
C INTO NPATH SUBSETS SO THAT EACH SUBSET IS EVALUATED ASYNCHRONOUSLY FROM
C OTHER SUBSETS. THE PRODCESSOR KEEPS ITERATING ON ITS SUBSET UNTIL ALL OF
C ITS COMPONENTS ARE CONVERGED. WHEN THIS SUBSET IS CONVERGED, ITS OWN FLAG
C IS SET TO 2 AND A TEST OF ALL THE FLAGS OF OTHER SUBSETS IS MADE. IF ANY
C OF THE SUBSETS HAS NOT BEEN CONVERGED YET THE CURRENT PROCESSOR ITERATES
C ON ITS SUBSET AGAIN AND SO ON UNTIL ALL THE SUBSETS ARE CONVERGED.
C THE SUBSETS ALLOCATE A NUMBER OF SEQUENTIAL LINES OF THE MESH AS IN
C PROGRAM A.1. ALSO HERE ONE SHARED ARRAY X AND TWO LOCAL ARRAYS HX AND AX
C ARE REQUIRED.

INTEGER*2 ITIME
DIMENSION X(26,26),HX(26,26),AX(26,26),ERROR(700),IFLAG(20)
ITIME(100),ITER(20)
$SHARED X,HX,AX,ERROR,IFLAG
$USEPAR

EPS=0.00001
MAXITR=1000

READ(5,90) N,NPATH,IPROC
90 FORMAT(12,I2,12,I2,12)
N=N-1
N2=N-2
WRITE(6,99) N2,MACITR,EPS,NPATH,IPROC
99 FORMAT(/'N2=',12,I2X,'MAXITR=',I4/2X,'EPS=',F10.6/2X,'NPATH=',I2/2X,'IPROC=',12)
DO 73 I=1,N
X(I,1)=100.0
DO 71 J=2,N
X(I,J)=0.0
71 CONTINUE
73 CONTINUE
NPeN2/NPATH
IFLAG(I)=1
CONTINUE

DOALL 1 IP=I.NPATH
THE START AND END POINTS OF SIZE OF THE SUBSET.
IPF=NP*IP+1
IPF=NP*IP+1
IF(IP,EQ,NPATH) IPF=N
C START ITERATIONS NOW UNTIL THIS SUBSET AND ALL OTHER SUBSETS HAVE BEEN
CONVERGED. first READ ALL THE COMPONENTS OF THE MESH BY A CRITICAL
SECTION.
CALL READST(HX)
C UPDATE THE ITERATION COUNTER ITER.
IM=IP+1+2
IF(IP,EQ,NPATH) IPF=N
 CALL TIMEST
C STARTTIMING THE ALGORITHM.
$SHARED X,N,N1,N2,ITER,NPATH,NP,MAXITR,EPS,IFLAG,TIME

IFERROR(700).IFLAG(20).
$USEPAR

EPS=0.00001
MAXITR=1000
READ(5,90) N,NPATH,IPROC
90 FORMAT(12,I2,12,I2,12)
N=N-1
N2=N-2
WRITE(6,99) N2,MACITR,EPS,NPATH,IPROC
99 FORMAT(/'N2=',12,I2X,'MAXITR=',I4/2X,'EPS=',F10.6/2X,'NPATH=',I2/2X,'IPROC=',12)
DO 73 I=1,N
X(I,1)=100.0
DO 71 J=2,N
X(I,J)=0.0
71 CONTINUE
73 CONTINUE
NPeN2/NPATH
IFLAG(I)=1
CONTINUE

DO 56 I=1,NPATH
ITER(I)=0
56 CONTINUE

DO 37 IPT=1,IPF
IM=(JP-1)*NP*N2+1
KL=IM
C EVALUATE THE SUBSET AND THE ERROR OF THE COMPONENTS.
DO 35 I=IP,IPF
DO 37 J=2,N
AX(I,J)=(HX(I-1,J)+HX(I,1,J)+HX(I,J-1)+HX(I,J+1))/4
ERROR(IM)=ABS(AX(I,J)-HX(I,J))/(1+ABS(HX(I,J)))
35 CONTINUE
37 CONTINUE
C A CRITICAL SECTION TO UPDATE THE CORRESPONDING LOCATIONS OF THE
C COMPONENTS OF THE SUBSET SO THAT OTHER PROCESSORS CAN USE THE NEW VALUES
CALL RELAES(IPS,IPF,AX,HX)
C TEST IF THE CURRENT SUBSET IS CONVERGED THEN SET ITS FLAG INTO 2. ELSE
C Iterate Again.
IF(ITER(IP),GT,MAXITR) GOTO 10
70 C TEST IF THE CURRENT SUBSET IS CONVERGED THEN SET ITS FLAG INTO 2. ELSE
C Iterate Again.
71 IFERROR(KL,GE,EPS) GOTO 45
72 IF(KL,EQ,KL) GOTO 75
74 IF(KL.LT,IM) GOTO 75
75 IFLAG(IP)=2
C A GLOBAL CONVERGENCE TEST TO TEST THE FLAGS OF OTHER SUBSETS. IF ANY OF
C THE SUBSETS HAS NOT BEEN CONVERGED YET THE CURRENT PROCESSOR ITERATES
C ON ITS SUBSET AGAIN AND SO ON UNTIL ALL THE SUBSETS ARE CONVERGED.
C THIS MEANS THAT THE PROCESSORS ARE WORKING ASYNCHRONOUSLY.
11
J1=1
30
77
IF(FLAG(J1).EQ.1) GOTO 45
81
J1=J1+1
82
IF(J1.LE.NPATH) GOTO 77
83
10
STOP
84
K4=1
85
29
IF(ITER(K4).GT.MAXITR) GOTO 10
86
K4=K4+1
87
IF(K4.LE.NPATH) GOTO 29
88
C END TIMING.
89
$DOALL 3
90
CALL TIMOUT(ITIME)
91
3
$SHARED
92
WRITE(6,87) ITIME
93
87
FORMAT(/2X,'TIMING'/8(16.2X))
94
DO 111 I=1,NPATH
95
WRITE(6,81) I
96
81
FORMAT(/2X,'CONVERGENCE IS ACHIEVED IN PROCESS NO.','12)
97
WRITE(6,89) ITER(I)
98
88
FORMAT(2X,'AFTER',1X,14,2X,'ITERATIONS')
99
111
CONTINUE
100
761
WRITE(6,890)
101
890
FORMAT(112X,'THE MESH IS')
102
DO 91 I=1,N
103
WRITE(6,44) (X(I,J),J=1,N)
104
44
FORMAT(/2X,7(F10.6,1X))
105
91
CONTINUE
106
GOTO 200
107
200
$STOP
108
$END

CRITICAL SECTION TO READ ALL THE COMPONENTS OF THE MESH AND PUT THEM
IN THE LOCAL ARRAY HY.

SUBROUTINE READST(HY)
DIMENSION X(26,26),HY(26,26)
$SHARED
REGION READST
ENTER READST
DO 23 K=1,N
DO 24 I=1,N
X(I,J)=HY(I,J)
CONTINUE
CONTINUE
DO 21 K=1,N
HY(I,J)=X(I,J)
CONTINUE
CONTINUE
EXIT READST
RETURN
END

A CRITICAL SECTION TO UPDATE THE COMPONENTS OF THE SUBSET AND PUT THEM
IN THE SHARED ARRAY X, THEN READ ALL COMPONENTS OF THE MESH FOR NEXT
ITERATION.

SUBROUTINE RELEAS(H1,I2,AY,HJ)
DIMENSION X(26,26),AY(26,26),HJ(26,26)
$SHARED
REGION RELEASE
ENTER RELEASE
DO 20 I=11,12
DO 142 J=2,N1
142
CONTINUE
CONTINUE
EXIT RELEASE
RETURN
END
PROGRAM A.3

C THIS PROGRAM IMPLEMENTS THE ASYNCHRONOUS POINT GAUSS-SEIDEL ITERATIVE
METHODOLOGY TO SOLVE THE 2-DIMENSIONAL DIRICHLET PROBLEM. THE LINES OF THE
MESH ARE PARTITIONED INTO SUBSETS USING THE SEQUENTIAL DECOMPOSITION.
THE PROCESSORS ASSIGNED TO SUBSETS WORK ASYNCHRONEOUSLY; I.E. EACH
PROCESSOR ITERATES ON ITS OWN SUBSET UNTIL ITS OWN SUBSET AND THE OTHER
SUBSETS ARE CONVERGED. A FLAG IS NEEDED TO CHECK FOR CONVERGENCE AS IN
PROGRAM A.2. HERE, ONLY ONE SHARED AND ONE LOCAL ARRAYS XN, XO ARE NEEDED.

C THE ALGORITHM CAN RUN FOR ANY NUMBER OF PATHS LESS THAN OR EQUAL TO THE
C NUMBER OF PROCESSORS. EACH PROCESSOR HAS ITS OWN ITERATION COUNTER/FLAG.

INTEGER*2 ITIME
DIMENSION XN(26,26), XO(26,26), ERR(100), ITER(10), ITIME(100)
*IFLAG(10)

$shared XN,N,NSIZE,NP,ITER,MAXITR,NPATH,EPSLON,IFLAG

C DEFINE THE REQUIRED ACCURACY.
EPSLON=0.0001
C READ NUMBER OF PATHS AND PROCESSORS AND THE NUMBER OF THE LINES IN THE
C MESH.
READ(5,90) N,NPATH,IPROC,PRINT
FORMAT(4(I2,1X))
NS=N-2
WRITE(6,95)
FORMAT(/:MAXITR=;I4/2X.;EPSLON=;FIO.o/2X.;N=;I2/2X.;NPATH=;I2
1/2X.;IPROC=;IZ)
C INITIALISE THE MESH COMPONENTS AND THE BOUNDARY CONDITIONS.
DO 11 I=1,N
XN(I,1)=100.0
DO 22 J=2,N
XN(I,J)=0.0
CONTINUE
C START TIMING THE ALGORITHM.
SDOALL
CALL TIMEST
C START THE ITERATIONS NOW FOR EACH PATH CARRIED OUT BY A PROCESSOR. THE
PROCESSOR STOPS ITERATING IF THE SUBSET IS CONVERGED.
DO 15 IP=1,NPATH
IPS=NP*(IP-1)+2
IPF=NP*IP+1
C READ ALL OF THE COMPONENTS OF THE MESH BY A CRITICAL SECTION AND
PUT THEM IN THE LOCAL ARRAY XO.
CALL READST(XO)
C UPDATE THE ITERATION COUNTER AND ITERATE FROM THIS POINT UNTIL ALL THE
COMPONENTS ARE CONVERGED.
ITER(IP)=ITER(IP)+1
IM=(IP-1)*NP*NS+l
KL-IM
C CALCULATE THE NEW VALUES OF THE COMPONENTS OF THIS SUBSET AND PUT THEM
IN THE ARRAY XO.
DO 24 I=IPS,IPF
DO 34 J=2,NSIZE
XO(I,J)=(XO(I,J+1)+XO(I,J-1)+XO(I+1,J)+XO(I-1,J))/4
ERR(IM)=ABS(XO(I,J)-XNU,,J)/(/IABS(XN(I,J))
IM=IM+l
CONTINUE
DO 24 CONTINUE
C UPDATE THE LOCATIONS IN THE SHARED ARRAY XN WHICH CORRESPOND TO THE
COMPONENTS OF THIS SUBSET. THIS IS DONE IN A CRITICAL SECTION SO THAT
OTHER PROCESSORS CAN USE THE NEW VALUES IF THEY REQUIRE THEM.
CALL RELEASE(IPS,IPF,XO)
IF(ITER(IP).GT.MAXITR) GOTO 15
C CHECK WHETHER ALL COMPONENTS OF THIS SUBSET ARE CONVERGED. IF YES, SET
THE FLAG INTO 2. OTHERWISE ITERATE AGAIN UNTIL THE CONVERGENCE TEST IS
SATISFIED.
IF(ERR(KL).GE.EPSLON) GOTO 50
KL=KL+l
IF(KL.LT.N) GOTO 76
C CHECK IF ALL SUBSETS ARE CONVERGED. IF ANY ONE IS NOT THEN THIS CURRENT C SUBSET ITERATES AGAIN UNTIL ALL SUBSETS SATISFY THE CONVERGENCE TEST.

IF(ILAG(L4).EQ.1) GOTO 50

L4=L4+1

IF(L4.LE.NPATH) GOTO 759

K4=K4+1

IF(K4.LE.NPATH) GOTO 29

C CHECK TO WRITE IF IT IS CONVERGENT OR DIVERGENT CASE.

IF(ITER(K4).GT.MAXITR) GOTO 200

K4=K4+1

IF(K4.LE.NPATH) GOTO 759

WRITE(6.80)

FORMAT(//2X.'THE TIMING')

DO 5 S5 1-1.NPATH

WRITE(6.64)

FORMAT(112X.'THE SOLUTION MESH IS')

CONTINUE

DO 45 J=2,NSIZE

CONTINUE

CONTINUE

EXIT READST

RETURN

END

C A CRITICAL SECTION TO READ ALL COMPONENTS OF THE MESH XN AND PUT THEM IN C THE LOCAL ARRAY XD.

SUBROUTINE READST(YO)

DIMENSION XN(26,26),YO(26,26)

*SHARED XN,N

*REGION READST

DO 13 I=1,N

DO 14 J=1,N

YO(I,J)=XN(I,J)

CONTINUE

CONTINUE

EXIT READST

RETURN

END

C A CRITICAL SECTION TO UPDATE THE COMPONENTS OF THE SUBSET AND PUT THEM C IN THE SHARED ARRAY XN SO OTHER PROCESSORS CAN USE THEM. THEN READ ALL C COMPONENTS OF THE MESH FOR NEXT ITERATION STEP.

SUBROUTINE RELEASES(11,12,Y0)

DIMENSION XN(26,26),YO(26,26)

*SHARED XN,N,NSIZE

*REGION RELEASES

*ENTER RELEASES

DO 34 I=1,12

DO 35 J=1,NSIZE

XN(I,J)=YO(I,J)

CONTINUE

CONTINUE

EXIT RELEASES

RETURN

END
PROGRAM A.4

C THIS PROGRAM IMPLEMENTS THE PURELY ASYNCHRONOUS ITERATIVE METHOD, WHERE
C NO SYNCHRONISATION IS REQUIRED. THE NPATH SUBSETS WORK ASYNCHRONOUSLY.
C SO THAT THE COMPONENTS OF THE SUBSET ARE UPDATED AS SOON AS THEY ARE
C EVALUATED. HENCE, ONLY ONE SHARED ARRAY XN IS REQUIRED IN THIS CASE.
C
INTEGER*2 ITIME
DIMENSION XN(26,26),IFLAG(20),ERROR(900),ITER(20),ITIME(100)
$SHARED XN,NS,NP,NSIZE,NPATH,EPSLON,ITER,MAXITR,IFLAG,ITIME
$USEPAR

MAXITR=1000
READ(5,90) N,NPATH,IPROC,EPSLON
90 FORMAT(3UZ,lX).1.X.F7.5)
C N IS THE NUMBER OF LINES (ROWS) IN THE MESH INCLUDING THE TWO LINES
C PRESENTING THE BOUNDARY. NS IS THE NUMBER OF ROW THAT ARE TO BE SOLVED.
NS=NSIZE-N
NP=NS/NPATH
WRITEC3,95) NS.MAXITR.EPSLON.NPATH,IPROC
95 FORMATC/'NS=',I4/'MAXITR=', 14/'EPSLON='F9.6/'NPATH=',I/2/'IPROC='.
C INITIALISE THE COMPONENTS AND THE BOUNDARY CONDITIONS OF THE MESH.
DO 11 I=1,N
XN(I,1)=100.0
DO 12 J=2,N
XN(I,J)=0.0
12 CONTINUE
DO 11 CONTINUE
DO 21 I=1,NPATH
ITER(I)=0
IM=(I-1)*NS*NP+1
K=IM
C EVALUATE THE COMPONENTS OF THE SUBSET BY USING THE POINT GAUSS-SEIDEL
C METHOD BUT THE NEW VALUES ARE IMMEDIATELY RELEASED TO THE SHARED
C ARRAY SO THAT ALL OTHER PROCESSORS CAN USE THEM.
DO 24 IP=1,NPATH
ERROR(IM)=ABS(XN(I,1)-XOLD)/(1+ABS(XOLD))
IM=IM+1
24 CONTINUE
IF(ITER(IP).GT.MAXITR) GOTO 15
IF(AL(0).GE.EPSLON) GOTO 50
IF(ITER(IP).GT.MAXITR) GOTO 210
K4=K4+1
IF(K4.LE.NPATH) GOTO 29
IF(ITER(K4).GT.MAXITR) GOTO 210
K4=K4+1
IF(K4.LE.NPATH) GOTO 29
C END TIMING.
$DOALL 4
CALL TIMOUT(ITIME)
$USEPAR
80   WRITE(6,89) ITIME
81   89 FORMAT(12X,'THE TIMING'/6(I6,2X))
82   DO 201 I=1,NPATH
83   WRITE(6,81) I
84   81 FORMAT(12X,'CONVERGENCE IS ACHIEVED IN PROCESS NO.',12)
85   WRITE(6,85) ITER(I)
86   85 FORMAT(12X,'AFTER',1X,I4,2X,'ITERATIONS')
87   201 CONTINUE
88   754 WRITE(6,754)
89   754 FORMAT(12X,'THE MESH COMPONENTS ARE')
90   DO 311 I=1,N
91   WRITE(6,65) (XN(I,J),J=1,N)
92   65 FORMAT(12X,7(F10.6,1X))
93   311 CONTINUE
94   GOTO 200
95   210 $DOALL 7
96   CALL TIMOUT(ITIME)
97    7 PARENT
98   WRITE(6,89) ITIME
99   WRITE(6,57)
100   57 FORMAT(12X,'NO CONVERGENCE IS ACHIEVED')
101  200 GOTO 766
102  200 $STOP
103   $END
PROGRAM A.5

THIS PROGRAM IMPLEMENTS THE 4-POINT BLOCK ITERATIVE METHOD WHERE THE SOR ITERATIVE METHOD IS USED. THIS IS AN ASYNCHRONOUS ALGORITHM TO SOLVE THE 2-DIMENSIONAL DIRICHLET PROBLEM. THE LINES OF THE MESH TO BE SOLVED ARE PARTITIONED INTO NPATH SUBSETS SO THAT EACH SUBSET IS ASSIGNED TO A GROUP OF SEQUENTIAL LINES. THE BLOCKS OF 4 ARE EVALUATED IN THE NATURAL ORDERING WHERE EACH TWO LINES WHICH FORM THE BLOCKS ARE TAKEN AT A TIME. THIS IS THE PROGRAM OF PROG 5.2 OF CHAPTER 5.

INTEGER*2 ITIME
DIMENSION XN(26,26).ERR(1300).ITER(6).IFLAG(6).ITIME(IOO)
SHAREO XN.N.N1.N2.NP.ITER.IFLAG.NPATH.EPS.MAXITR.W.ITER
USEPAR
MAXITR=1000
EPS=0.000001
READ(S.90) N,NPATH,IPRINT,W,Wl
90 FORMAT(I2.1X.12.IX.F4.2.lX.F4.2)
N2=N-2
Nl=N-1
NP=N2/NPATH
IF(NP.EQ.1)
NP=1
WRITE(0.95) MAXITR,EPS,N2,NPATH
95 FORMAT('/'MAXITR='.I4/2X,'EPS='F10.6/2X,'N2='.12/2X.'NPATH=''.I2)
C RUN THE ALGORITHM WITH W WHICH IS INCREASED EACH STEP BY Wl IN ORDER TO FIND THE EXACT RELAXATION FACTOR.
DO 37 II=1,10
W=W+Wl
WRITE(6.S64)
864 FORMAT(/2X.'W=',F7.3)
C INITIALISE THE COMPONENTS AND THE BOUNDARY OF THE MESH.
DO 11 II=N-1
XNCI.I,I)=100.0
DO 22 II=2,N
XN(II,j)=0.0
22 CONTINUE
DO 17 I=NPATH
ITER(I)=0
IFLAQ(I)=1
17 CONTINUE
C START TIMING HERE
DOALL 1
CALL TIMEST
PAREND 1
C SET UP A PROCESS FOR EACH PROCESSOR TO ITERATE ASYNCHRONOUSLY.
SDOPAR 15
IP=IP+NP-1)+2
IPF=NP*IP+1
C INITIALISE AN INDEX.IND.FOR THE ERROR VECTOR IN EACH PROCESS AND TO ITERATE ON THE SUBSET AS IN THE PURELY ASYNCHRONOUS ALGORITHM. ITERATE ON THESE COMPONENTS UNTIL ALL OF THEM ARE CONVERGED AS WELL AS OTHER SUBSETS CARRIED OUT BY OTHER PROCESSORS ARE CONVERGED.
ITER(IP)=ITER(IP)+1
INO=(IP-1)*N2+1
KL=IND
C PICK EACH TWO CONSEQUENT LINES AT A TIME SO THAT THE BLOCKS TO BE EVALUATED CAN BE SOLVED AS COMPLETE.
DO 35 J=2,N2-2
35 XOLD=XNCI.I,J)
XOLD=XN(II,J)+XNCI.I+1,J)
S2=XN(II,J+1)+XN(I,J+2)
S3=XN(II+1,J)+XN(I+1,J+1)
S4=XN(II+1,J-1)+XN(I,J-2)
SS=S1+S2+S3+S4
S6=S2+S4+S4
XNEW=(7*S1+56+S3)/24
C EVALUATE THE COMPONENTS WITH W AND PUT THE NEW VALUE IN SHARED ARRAY XN.
XNEW=W*(XNEW-XOLD)+XOLD
ERRIND=ABS(XNEW-XOLD)/(I+ABS(XOLD))
\[
\begin{align*}
X_{\text{NEW}}(I, J+1) &= X_{\text{NEW}} \\
\text{IND} &= \text{IND} + 1 \\
X_{\text{OLD}} &= X_{\text{NEW}}(I, J+1) \\
X_{\text{NEW}} &= (7S_3 + S_6 + S_1) / 24 \\
\text{ERR}(\text{IND}) &= \frac{\text{ABS}(X_{\text{NEW}} - X_{\text{OLD}})}{1 + \text{ABS}(X_{\text{OLD}})} \\
X_{\text{NEW}} &= W \cdot (X_{\text{NEW}} - X_{\text{OLD}}) + X_{\text{OLD}} \\
X_{\text{NEW}}(I, J+1) &= X_{\text{NEW}} \\
\text{IND} &= \text{IND} + 1 \\
X_{\text{OLD}} &= X_{\text{NEW}}(I, J+1) \\
X_{\text{NEW}} &= (7S_4 + S_5 + S_2) / 24 \\
\text{ERR}(\text{IND}) &= \frac{\text{ABS}(X_{\text{NEW}} - X_{\text{OLD}})}{1 + \text{ABS}(X_{\text{OLD}})} \\
\text{IND} &= \text{IND} + 1 \\
\text{END OF FIRST LOOP OF } J \\
\text{END}
\end{align*}
\]
PROGRAM A

THIS PROGRAM IMPLEMENTS THE 4-POINT BLOCK ITERATIVE METHOD THAT USES THE SOR METHOD. THIS PROGRAM IMPLEMENTS PROG S OF CHAPTER S.

THIS IS ASYNCHRONOUS PROGRAM—EACH PROCESSOR EVALUATES A NUMBER OF BLOCKS WHICH ARE TAKEN CORRESPONDING TO THE MODIFICATION OF THE SHARED INDEX SO THAT EVERY PROCESSOR EVALUATES SOME BLOCKS BUT NOT GROUPED IN SUBSET ALLOCATED PERMANENTLY.

THE BLOCKS ARE ARRANGED IN THE SHARED LIST SO THAT THEY CAN BE TAKEN IN A RED-BLACK ORDERING.

INTEGER*2 ITIME
DIMENSION XN(26,26),ERRCI0),LIST(2,1026),IPCNV(64),ITIMECI00)

USEPAR

SET THE MAX LIMIT FOR ITERATIONS AND THE REQUIRED ACCURACY.
MAXITR=1000
EPS=0.00001
N,NPATH,IPRIN,T,W,Wl
READCS,90)
90 FORMAT(3(I2.1X).2(F4.2.1X)
N2 IS THE NUMBER OF LINES IN THE MESH AFTER SUBTRACTING THE 2 LINES OF THE BOUNDARY.
N2=N-2
N1=N-1
NP=(N2*N2)/4
WRITE(6.9S) MAXITR,EPS,N2,NPATH,NP
9S FORMATC/'MAXITR=',14/2X'.EPS-',FI0.6/2X,'N2=',I2/2X.'NPATH=',I2/2X.
NO OF BLOCKS NP=',I3)

RUN THE ALGORITHM WITH W BEING INCINENTED BY Wl TO OBTAIN THE EXACT W
DO 37 IW=1,10
W=W+Wl
WRITE(6.864) W
664 FORMATC/2X,'W z ',F7.3)

INITIALISE THE MESH COMPONENTS AND THE BOUNDARY CONDITIONS.
D0 11 I=1,N
XM(I,I)=100.0
D0 22 J=2,N
XN(I,J)=0.0
CONTINUE

IND=0
DO 27 Il=2,N2,2
Kl=6-Kl
DO 29 Kl=2,N2,2
LIST(l,IND)=Il
LIST(2,IND)=J
IND=IND+1
CONTINUE

INITIALISE LIND, FOR OBTAINING THE CORRESPONDING INDEX TO THE LIST TO KNOW THE INDICES OF EACH BLOCK.
K=1
C INITIALISE IND, FOR THE NEXT DO LOOP
IND=1
I=0
K=4
135 DO 27 Il=2,N2.2
Kl=6-Kl
DO 28 Jl=Kl,N2.4
LIST(l,IND)=Il
LIST(2,IND)=Jl
IND=IND+1
CONTINUE

CONTINUE

IF(IN(GT.1) GOTO 1133
Kl=2
GOTO 135
C INITIALISE A GLOBAL CONVERGENCE FLAG
1133 IFLAG=1
C START TIMING HERE
D0 1 #DALL 1
72 CALL TImEST
1 #PEND
C SET UP A PROCESS FOR EACH PROCESSOR
$OPAR IS IP=1,NPATH
C NOW EACH PATH UPDATES THE INDEX TO THE LIST TO GET THE ROW AND COLUMN NUMBERS OF THE BLOCK. A CRITICAL SECTION IS REQUIRED HERE.
IF(IFDEF.EQ.O) GOTO 15
CALL UPDATE(ILOC)
1=LIST(1,ILOC)
J=LIST(2,ILOC)
C SOLVE NOW THE BLOCK WHOSE INDECES ARE i, j BY EVALUATING EACH OF ITS OWN

C COMPONENTS.

KK=1
IS=KK
XOLD=XN(I,J)
S1=XN(I,J+1)+XN(I-1,J)
S2=XN(I+1,J+1)+XN(I+1,J)
S4=XN(I+1,J)+XN(I,J+1)
S5=S1+S3+S4+S5
S6=(7*S4+S6+S4)/24
XNEW=(7*S2+S5+S1)/24
XNEW=W*(XNEW-XOLD)+XOLD
ERRK=ABS(XNEW-XOLD)/(1+ABS(XOLD))

C USE THE RELAXATION FACTOR WITH EACH COMPONENT OF THE BLOCK. THE NEW VALUE
C OBTAINED IS ASSIGNED TO THE SHARED ARRAY XN IMMEDIATELY.

KK=KK+1
XOLD=XN(I+1,J+1)
S1=XN(I,J+1)+XN(I,J+1)
S2=XN(I,J)+XN(I,J+1)
S4=XN(I+1,J)+XN(I+1,J)
S5=XN(I,J)+XN(I,J+1)
S6=(7*S4+S6+S1)/24
XNEW=(7*S2+S5+S4)/24
XNEW=W*(XNEW-XOLD)+XOLD
ERRK=ABS(XNEW-XOLD)/(1+ABS(XOLD))

C SET FLAG OF THE CURRENT PROCESSOR TO 1 IF ANY COMPONENT OF THE BLOCK IS
C NOT CONVERGED ELSE SET IT TO 2.

IF(ERRK.LE.EPS) GOTO 79
IPCNV(ILOC)=1
GOTO 541
ICONTINUE
IFLAG=0
DO 79 IL=1,N
IF(IPCNV(1L).EQ.O) GOTO 79
CONTINUE
GOTO 50
CONTINUE

C CHECK IF ALL COMPONENTS OF ALL BLOCKS ARE CONVERGED. IF ANY COMPONENT
C HAS NOT BEEN CONVERGED YET LOOP AGAIN TO EVALUATE ANOTHER BLOCK. THIS
C MEANS THAT THE PROCESSORS WORK ASYNCHRONOUSLY.

IF(IPCNV(ILOC).EQ.O) GOTO 67
IFLAG=1
GOTO 50
CONTINUE
GOTO 67
IFLAG=0
CONTINUE

C CHECK FOR DIVERGENCY

IF(IPCNV(ILOC).EQ.O) GOTO 67
WRITE(6,417) ITER2
WRITE(6,418) ITER2
417 FORMAT('CONVERGENCE HAS ACHIEVED AFTER',1X,4,1X, 'ITERATIONS')
418 FORMAT('THE SOLUTION MESH IS')
150 154 IF(IPRINT.GT.0) GOTO 37
150 154 WRITE(6,64)
151 64 FORMAT('THE SOLUTION MESH IS')
152 64 DO 810 J=1,N
153 65 WRITE(6,65) (XN(I,J,J),J=1,N)
154 65 FORMAT((1X,7(F10.6,1X))
155 65 CONTINUE
156 65 GOTO 37
157 250 WRITE(6,61)
158 61 FORMAT('NO CONVERGENCE IS ACHIEVED')
C A CRITICAL SECTION TO UPDATE THE SHARED INDEX TO THE LIST OF BLOCKS.

SUBROUTINE UPDATE(ILOC)
DIMENSION XN(35,35)

REGION UPDATE
ENTER UPDATE
IF(LIND.EQ.NP) LIND=0
LIND=LIND+1
ILOC=LIND
EXIT UPDATE
RETURN
END
PROGRAM A.7

THIS PROGRAM IMPLEMENTS THE BUBBLE SORT AND THE 2-WAY MERGE ALGORITHMS.

C THE INPUT SET TO BE SORTED IS PARTITIONED INTO NPATHS SUBSETS WHICH ARE
C EQUAL TO OR GREATER THAN THE NUMBER OF PROCESSORS USED IN THE SORTING
C PROCEDURE. EACH PROCESSOR SORTS ITS OWN SUBSET USING THE SEQUENTIAL
C BUBBLE SORT WHICH IS OF ORDER N^2. THEN, THE SUBSETS WILL BE SORTED
C WITHIN THEMSELVES BUT NOT AMONG THEM EACH OTHER. Thus, THE MERGE PROCEDURE
C IS TO MERGE THESE SUBETS TO OBTAIN THE FINAL SORTED SET. TWO ARRAYS ARE
C REQUIRED ONE IN THE SHARED AND ONE IN THE LOCAL MEMORY.

INTEGER*2 ITIME

DIMENSION A(2100),C(2100),ITIME(100)

USEPAR

READ THE DATA SIZE, NUMBER OF PATHS, PROCESSORS, AND LOG(NPATHS).

READ(88) N,NPATHS,NPROC,LOGP,IPRINT

FORMAT(5(14.1X)) WRITE(b.99)

do 1 b=1,N

A(I)=RANF(DUMMY)

IF(IPRINT.LT.b) WRITE(6.95)

C FIND THE SIZE OF EACH SUBSET TO BE ASSIGNED TO A PATH.

NELM=N/NPATHS

C START TIMING OF THE SORT PROCEDURE.

$DOALL 3

CALL RANSET(2)

DO 2 I=b,N

2 A(I)=RANF(DUMMY)

IF(IPRINT.LT.I) WRITE(6.95)

IF(IP.EQ.0) IE=0

CALL RANSET(2)

do 30 J=1,b,IE

30 CONTINUE

IF(JL.EQ.0) GOTO 104

CALL TIMEST

CALL PAREND

IF(LOOP.NO) GOTO 110

C END TIMING THE SORT PROCEDURE.

$DOALL 5

CALL TIMOUT(ITIME)

CALL PAREND

WRITE(6,95) (A(I),I=I,N)

WRITE(6,555) FORMAT(1'THE TIME FOR SORTING ALONE')

WRITE(6,112) ITIME

C START TIMING THE MERGE PROCEDURE.

$DOALL 6

CALL TIMEST

C THE 2-WAY MERGE REQUIRES LOG(NPATHS) SEQUENTIAL STEPS, BUT IN EACH STEP A
C NUMBER OF PARALLEL PATHS ARE GENERATED WHERE EACH PATH MERGES 2 SUBSETS
C DO 100 KP=1,LOGP

C OBTAIN THE NUMBER OF PATHS TO BE CARRIED OUT IN PARALLEL.

NP=NPATHS/2**KP

C ASSIGN A PROCESSOR FOR EACH PATH.

$DOALL 13

C CALCULATE THE START AND END POINTS OF THE CURRENT 2 SUBSETS TO BE MERGED

C SINCE EACH PATH HERE MERGES 2 SUBSETS ONLY, A LOCAL VECTOR C IS USED TO
C HOLD THE PARTIAL RESULTS OF THE MERGE.

IPS=2**KP*NELM*(IP-1)+1

JPS=IPS+NELM*(2**KP-1)

J=1

$DOALL 119

WRITE(6,112) ITIME
LI=IPS
L2=J1
I2=IPS-1
22 I2=I2+1
IF(A(L1),LT,A(L2)) GOTO 70
C(I2)=A(L2)
L2=L2+1
IF(L2.LE.J2) GOTO 22
GOTO 37
70 C(I2)=A(L1)
L1=L1+1
IF(L1.LE.IPE) GOTO 22
GOTO 24
C IN CASE SOME ELEMENTS OF THE FIRST SUBSET ARE GREATER THAN THAT IN THE
SECOND SUBSET THEN MOVE ALL THESE ELEMENTS INTO THE VECTOR C WITHOUT
ANY COMPARISON.
37 DO 16 I3=LI,IPE
I2=I2+1
C(I2)=A(I3)
16 CONTINUE
GOTO 80
C IN CASE SOME ELEMENTS OF THE SECOND SUBSET ARE GREATER THAN THAT OF THE
FIRST SUBSET THEN MOVE THESE ELEMENTS TO THE VECTOR C WITHOUT COMPARISON
24 DO 43 I4=L2,J2
I2=I2+1
C(I2)=A(I4)
43 CONTINUE
C WHEN THE MERGE IS COMPLETE MOVE THE ELEMENTS FROM THE LOCAL VECTOR C TO
THE SHARED VECTOR A.
12 A(K1)=C(K1)
15 CONTINUE
C WHEN ALL SUBSETS ARE PARTIALLY MERGED IN THIS STEP DO THE NEXT STEP OF
THE ALGORITHM.
100 CONTINUE
C END THE MERGE TIMING.
110 $DOALL 65
116 CALL TIMEOUT(ITIME)
119 $STOP
120 WRITE(6,443)
121 FORMAT(’THE TIME FOR MERGING ALONE’)
122 WRITE(6,112) ITIME
123 FORMAT(1X,8(E12.5,2X))
124 IF(IPRINT.GT.1) GOTO 156
125 WRITE(6,95) (A(I),I=1,N)
126 95 FORMAT(1X,5(E12.5,2X))
127 156 $STOP
128 $END
PROGRAM A.8

C THIS PROGRAM IMPLEMENTS THE MERGE PROCEDURE PROPOSED BY BAUDET
C THAT CORRESPONES TO THE ODD-EVEN REDUCTION. THE SORT PROCEDURE IS
C THE NEIGHBOURS SORT IMPLEMENTED BY BAUDET & STEVENSON.
C SORTED IS PARTITIONED INTO NPATHS SUBSETS SO THAT THEY WILL BE SORTED
C PARTLY. EACH SUBSET IS SORTED SUCH AS EACH TWO NEIGHBOURING ELEMENTS ARE
C COMPARED THEN THE RESULTANT SMALL SUBSETS ARE COMPARED IN THE SAME WAY
C TO GET SUBSETS OF DOUBLED SIZE. THIS CONTINUES UNTIL THE FINAL STEP
C THAT MERGES ONLY TWO RESULTANT SUBSETS. THIS IS SIMILAR TO THE 2-WAY
C MERGE. THEN THE ORIGINAL SUBSET WILL BE SORTED AT THE END. THE MERGE
C PROCEDURE IS REQUIRED TO MERGE THE SUBSETS TO GET THE FINAL SORTED SET.
C THIS MERGE REQUIRES NPATHS STEPS. IN THE ODD-NUMBERED STEPS, THE ODD-
C NUMBERED SUBSETS ARE MERGED WITH THE EVEN-NUMBERED ONE. IN THE EVEN-
C NUMBERED STEP THE EVEN-NUMBERED SUBSETS ARE MERGED WITH THE ODD-NUMBERED
C ONE. TWO VECTORS ARE REQUIRED SHARED AND LOCAL VECTORS.

 INTEGER*2 ITIME
 DIMENSION A(2100),C(2100),ITIME(100)
 $SHARED N,NPATHS,A,K,NELM,MP,ITIME

 READ THE DATA SIZE, NUMBER OF PATHS AND NUMBER OF PROCESSORS TO BE USED.

 READ(S,SS) N,NPATHS,NPROC,IPRINT

 es
 FORMAT(4(I4,IX))

 WRITE(6,99) N,NPATHS,NPROC

 99 FORMAT(/'Ns',14/'NPATHS=',14/'NPROC=',14)

 C GENERATE THE NUMBERS TO BE SORTED RANDOMLY BY THE LIBRARY ROUTINES RANF

 C AND RANSET.

 CALL RANSET(2)

 DO
 A(I)=RANF(DUMMY)
 IF(IPRINT.OT.I) OOTO 113
 WRITE(6,95) (A(I), I-1, N)
 113 NELH=N/NPATHS

 C START TIMING FOR THE SORT PROCEDURE.

 $DOALL
 CALL TIMEST
 $PAREND

 C ASSIGN A PATH FOR EACH SUBSET.

 $DOPAR 10 IPROC=I,NPATHS
 NV=NELM-I
 DO
 II=1,NV,II
 IS1=NELM*(IPROC-I)+IP
 IE1=NELM*(IPROC-I)+IP-1+II
 IS2=IS1+II
 IE2=IE1+II
 IL=IS1
 L2=IS2
 ID=ID-1
 IF(IL.LE.IE2) GOTO 66
 IF(Il.LE.IE1) GOTO 42
 C(ID)=A(IL)
 C(ID)=A(IL+1)
 CONTINUE
 CONTINUE
 CONTINUE

 IF(IL.LE.IE2) GOTO 42
 DO 48 14=L2,IE2
 ID=ID+1
 C(ID)=A(IL)
 48 CONTINUE

 IF(IL.LE.IE1) GOTO 42
 DO 48 14=L1,IE1
 ID=ID+1
 C(ID)=A(IL)
 48 CONTINUE

 C END TIMING.

 CALL TIMOUT(ITIME)

 $DOALL 3
 CALL TIMEST

 $PAREND
81 WRITE(6,555)
82 555 FORMAT('THE TIME FOR SORTING ALONE')
83 WRITE(6,112) ITIME
84 C START MERGE TIMING.
85 $DOALL 6
86 CALL TİMEST
87 6 $PAREND
88 MP=1
89 C NPATHS SEQUENTIAL STEPS ARE REQUIRED TO COMPLETE THE MERGE.
90 DO 100 K=1,NPATHS
91 IF(K.EQ.NPATHS.AND.NPATHS.EQ.2) GOTO 100
92 IF(K.EQ.MP) GOTO 13
93 C TO CHOOSE FOR ODD OR EVEN STEPS SUBSETS.
94 NF=NPATHS/2-1
95 MP=MP+2
96 GOTO 17
97 13 NF=NPATHS/2
98 C C GENERATE NP PATHS WHICH IS HALF OF THE NUMBER OF SUBSETS OBTAINED FROM
99 C THE SORTING PART. THEREFORE, EACH PATH HEROES 2 SUBSETS.
100 C 2-WAY MERGE OF PROGRAM A.7.
101 17 $DOPAR 15 IH=1,NP
102 C GET THE START AND END POINTS OF THE TWO SUBSETS AND MERGE THEM IN THE
103 C 2-WAY MERGE OF PROGRAM A.7.
104 IPS=2*NELM*(IH-1)+(MP-K)*NELM+1
105 IPE=NELM*(2*IH-1)+(MP-K)*NELM
106 J1=IPS+NELM
107 J2=IPE+NELM
108 LI=IPS
109 L2=J1
110 I2=IPS-1
111 22 I2=I2+1
112 IF(A(L1).LT.A(L2)) GOTO 70
113 C(I2)=A(L2)
114 L2=L2+1
115 IF(L2.LE.J2) GOTO 22
116 GOTO 37
117 70 C(I2)=A(L1)
118 LI=LI+1
119 IF(LI.LE.IPE) GOTO 22
120 GOTO 24
121 37 DO 13=LI.IPE
122 13=LI+1
123 C(I2)=A(I3)
124 16 CONTINUE
125 GOTO 80
126 24 DO 43 I4=L2.J2
127 43 I4=I4+1
128 C(I2)=A(I4)
129 43 CONTINUE
130 C PUT THE SORTED ELEMENTS BACK INTO THE SHARED VECTOR A.
131 80 DO 12 K1=IPS.J2
132 12 A(K1)=C(K1)
133 15 $PAREND
134 C DO NOW THE NEXT STEP OF THE MERGE ALGORITHM.
135 100 CONTINUE
136 C END MERGE TIMING.
137 110 $DOALL 55
138 CALL TIMOUT(ITIME)
139 55 $PAREND
140 WRITE(6,443)
141 443 FORMAT("THE TIME FOR MERGING ALONE")
142 WRITE(6,112) ITIME
143 112 FORMAT(1X,B16.1X)
144 IF(IPRINT.GT.1) GOTO 156
145 WRITE(6,95) (A(I).I=1,N)
146 95 FORMAT(1X,S12.5,2X)
147 156 $STOP
148 $END
PROGRAM A.9

---

C THIS PROGRAM IMPLEMENTS THE PARALLEL MERGING BY THE JUMP SEARCHING
C ALGORITHM.TWO LEVELS OF JUMP ARE USED: FIRST JUMP SIZE AS (N**1/2)
C THE OTHER JUMP SIZE AS (N**1/4). IN THIS ALGORITHM A NUMBER OF SUBSETS
C ARE OBTAINED, AND THE MERGE PROCEEDS AS IN THE 2-WAY MERGE
C I.E., LOG(NPATHS) SEQUENTIAL STEPS ARE REQUIRED WHERE EACH STEP PERFORMS
C THE MERGE OF TWO SUBSETS IN PARALLEL WITH OTHER PATHS. BUT THE WORK
C INSIDE EACH PATH IS BY THE JUMP SEARCHING. THE SORT PROCEDURE TO SORT
C SUBSETS IS THE BUBBLE SORT DESCRIBED IN PROGRAM A.7. THE NUMBER OF PATHS
C CAN BE EQUAL TO OR GREATER THAN THE NUMBER OF PROCESSORS.TWO VECTORS ARE
C REQUIRED ONE IN THE SHARED AND ONE IN THE LOCAL MEMORY.

C READ THE DATA SIZE, NUMBER OF PATHS, NUMBER OF PROCESSORS, AND LOG(NPATHS).
C
READ(5,98) N,NPATHS,NPROC,LOGP,IPRINT

98 FORMAT(15,I4)

WRITE(6,99) N,NPATHS,NPROC,LOGP

99 FORMAT(/'N=',I4/'NPATHS=',I4/'NPROC=',I4/'LOGP=',I4)

C GENERATE THE DATA RANDOMLY BY THE RANDOM GENERATOR, RANSET, RANF WHICH
C ARE LIBRARY ROUTINES WORK IN THE INTERVAL [0,1].

CALL RANSET(2)

DO 11 J=1,N
   A(J)=RANF(DUMMY)
   IF(IPRINT.GT.1) GOTO 11

11 WRITE(6,44) J

44 FORMAT('J=',I4)

C START TIMING FOR THE SORTING PART.

C NPATHS SUBSETS TO BE SORTED IN PARALLEL BY THE BUBBLE SORT ALGORITHM.

DO ALL
   CALL TIMEST

C NPATHS SUBSETS TO BE SORTED IN PARALLEL BY THE BUBBLE SORT ALGORITHM.

DO 20 IP=1,NPATHS
   IS=NELM*(IP-1)+1
   IE=NELM*IP
   IF(IP.EQ.NPATHS) IE=N
   IBND=IE
   IE=IBND-I
   DO 30 J=IS,IE
      IF(A(J).LE.A(J+1)) GOTO 30
      T=A(J)
      A(J)=A(J+1)
      A(J+1)=T
   30 CONTINUE
   IF(JL.EQ.0) GOTO 10
   JL=JL+1
   DO 40 J=JL,IE
      IF(A(J).LE.A(J+1)) GOTO 30
      T=A(J)
      A(J)=A(J+1)
      A(J+1)=T
   40 CONTINUE

10 CALL TIMEST

C START THE MERGE TIMING.

C MERGE NOW BY USING THE JUMP SEARCH ALGORITHM. THIS REQUIRES
C LOG(NPATHS) SEQUENTIAL STEPS AS IN THE 2-WAY MERGE.A LOCAL VECTOR C
C IS USED TO HOLD THE PARTIAL RESULTS.

KP=1, LOOP

DO 110 KP=1,LOGP
   N=NPROC/2**KP

110 CONTINUE

C START MERGING NOW BY USING THE JUMP SEARCH ALGORITHM. THIS REQUIRES
C LOG(NPATHS) SEQUENTIAL STEPS AS IN THE 2-WAY MERGE.A LOCAL VECTOR C
C IS USED TO HOLD THE PARTIAL RESULTS.

DO 110 KP=1,LOGP
   N=NPATHS/2**KP

110 CONTINUE

C NPATHS TO BE CARRIED OUT IN PARALLEL WHERE EACH PROCESSOR MERGES TWO-
C SUBSETS TO GENERATE SUBSET OF DOUBLE SIZE.

DO 15 IP=1,NP
81 IS1=2*KP+NELM*(IP-1)+1
82 IE1=2*(KP-1)+NELM*(2*IP-1)
83 IS2=IS1+NELM*2*(KP-1)
84 IE2=IE1+NELM*2**(KP-1)
85 ID=IS1-1
86 KM=IS2
87 I2=IS2
88 X=FLOAT(JUMP1)
89 DO 20 I=IS1,IE1
90 ITWO=0
91 KF=JUMP1
92 IM=KM
93 I2=KM
94 ITWO=ITWO+1
95 IM=KM
96 IF(IM,GT,IE2) GOTO 217
97 DO 20 IM=K1,IE2
98 IC IM=K1
99 IF(IM.GT.IE2) GOTO 217
100 IM=K1
101 IC IM=K1
102 IF IM.HT.1E2) GOTO 217
103 IM=K1
104 CONTINUE
105 I2=KM
106 DO 20 IM=K1,IE2
107 IC IM=K1
108 IF IM.GT.1E2) GOTO 22
109 IM=IM+1
110 IF IM.GT.1E2) GOTO 217
111 IM=IM-1
112 IC IM=K1
113 IF IM.LT.1W) GOTO 319
114 ITWO=ITWO+1
115 IF ITWO.GT.2) GOTO 117
116 IF(ITWO.GT.2) GOTO 117
117 IF(IM.LE.1E2) GOTO 22
118 IM=IM+1
119 IF IM.GT.1W) GOTO 319
120 ID=ID+1
121 IC ID=AI(I)
122 ITWO=ITWO+1
123 IC ID=AI(I)
124 KM=IM+1
125 IF(KM.IE.1E2) GOTO 217
126 DO 11 IM=KM,1E2
127 IF IM.LT.AI(1)) GOTO 28
128 ID=ID+1
129 IC ID=AI(I)
130 GOTO 20
131 IC ID=AI(I)
132 IC ID=AI(I)
133 KM=KM+1
134 CONTINUE
135 IC ID=AI(I)
136 IC ID=AI(I)
137 IC ID=AI(I)
138 IC ID=AI(I)
139 IC ID=AI(I)
140 IC ID=AI(I)
141 IC ID=AI(I)
142 IC ID=AI(I)
143 IC ID=AI(I)
144 IC ID=AI(I)
145 IF ITWO.GT.2) GOTO 319
146 JUMP2=ifix(sort(XJ))
147 KF=JUMP2
148 IM=IM+1
149 IM=IM+1
150 IF IM.GT.1W) GOTO 319
151 IF IM.GT.1W) GOTO 22
152 IF IM.LT.AI(1)) GOTO 28
153 DO 23 14=IM,KM
154 IC ID=AI(I)
155 IC ID=AI(I)
156 KM=KM+1
157 CONTINUE
158 IC ID=AI(I)
159 IC ID=AI(I)
160 GOTO 20

C COMPARE THE ELEMENTS OF THE FIRST SUBSET WITH THE ELEMENT OF THE SECOND
C SUBSET WHICH IS ON POSITION (Z**1/2), WHERE Z IS THE SIZE OF THE SUBSET.
C IF THE ELEMENT IS GREATER THAN THAT IN SECOND SUBSET THEN ANOTHER JUMP
C IS MADE. IN THIS CASE THE JUMP IS OF SIZE(Z**1/4). IF THE ELEMENT IS
C STILL GREATER COMPARE IT WITH THE REST OF SECOND SUBSET USING THE SIMPLE
C SEARCH.
105 IF(A(I),EQ,A(IM)) GOTO 111
106 IF(A(I),LT,A(IM)) GOTO 62
107 IM=IM+1
108 IF(IM.LE.1E2) GOTO 22
109 IM=IM+1
110 IC IM=K1
111 IC IM=K1
112 IC IM=K1
113 IC IM=K1
114 IC IM=K1
115 IC IM=K1
116 IC IM=K1
117 IC IM=K1
118 IC IM=K1
119 IC IM=K1
120 IC IM=K1
121 IC IM=K1
122 IC IM=K1
123 IC IM=K1
124 IC IM=K1
125 IC IM=K1
126 IC IM=K1
127 IC IM=K1
128 IC IM=K1
129 IC IM=K1
130 IC IM=K1
131 IC IM=K1
132 IC IM=K1
133 IC IM=K1
134 IC IM=K1
135 IC IM=K1
136 IC IM=K1
137 IC IM=K1
138 IC IM=K1
139 IC IM=K1
140 IC IM=K1
141 IC IM=K1
142 IC IM=K1
143 IC IM=K1
144 IC IM=K1
145 IC IM=K1
146 IC IM=K1
147 IC IM=K1
148 IC IM=K1
149 IC IM=K1
150 IC IM=K1
151 IC IM=K1
152 IC IM=K1
153 IC IM=K1
154 IC IM=K1
155 IC IM=K1
156 IC IM=K1
157 CONTINUE
158 IC IM=K1
159 IC IM=K1
160 GOTO 20
IF(KM.GT.12) GOTO 25
DO 7 II=KM,I2
ID=ID+1
7 C(ID)=A(II)
KM=I2+1
II=I2+1
GOTO 107
II=KM
DO 107 II=II,IM
IF(A(I).GT.A(II)) GOTO 27
ID=ID+1
C(ID)=A(I)
KM=IM
GOTO 20
ID=ID+1
C(ID)=A(I)
KM=IM+1
CONTINUE
IF(KM.EQ.IM) GOTO 220
DO 39 I5=KM,IB
ID=ID+1
C(ID)=A(I5)
39 CONTINUE
IF(KM.GT.IE2) GOTO 80
DO 410 I=KM,IE2
ID=ID+1
C(ID)=A(I)
410 CONTINUE
C PUT THE ELEMENTS BACK IN THE SHARED VECTOR A.
DO 12 KI=IM,IE2
A(KI)=C(KI)
12 CONTINUE
END THE MERGE TIMING AND PRINT OUT THE OUTPUT SET.
CALL TIMEOUT(ITIME)
WRITE(6,443)
FORMAT(//'THE TIME FOR MERGING ALONE'\))
WRITE(6,112) ITIME
IF(IPRINT.GT.1) GOTO 156
WRITE(6,95) (A(I),I=1,N)
FORMAT(1X,5(E12.5,2X))
STOP
END
PROGRAM I.10
---
C THIS PROGRAM IMPLEMENTS THE PARALLEL QUICKSORT ALGORITHM. THE MEDIAN OF
C THREE ELEMENTS. FIRST THE ORIGINAL INPUT SET IS PARTITIONED TO OBTAIN
C TWO SUBSETS S1 & S2 SO THAT THE ELEMENTS OF S1 ARE LESS THAN THAT OF S2.
C THESE SUBSETS ARE KEPT IN THE QUEUE IF THEY ARE GREATER THAN M NUMBERS.
C THEN THE SUBSETS ARE POPPED OUT OF THE QUEUE TO BE PARTITIONED
C AGAIN AND SO ON UNTIL THERE IS NO SUBSET IN THE QUEUE. IN THIS CASE ALL
C SUBSETS ARE OF SIZE 1 WHICH CAN BE SORTED BY THE LINEAR INSERTION SORT.
C HERE, THE PARALLELISM IS INCREASED IN EACH STEP OF THE PARTITIONING, I.E.
C FIRST ONLY ONE PROCESSOR IS USED THEN TWO PROCESSORS ARE USED IN THE
C SECOND STEP AND SO ON THE NUMBER OF PROCESSORS IS DOUBLED IN EACH STEP
C OF THE PARTITIONING, THE INPUT SET IS KEPT IN THE SHARED MEMORY. I.H.
C SETS ARE REQUIRED FOR THE QUEUE, ONE TO HOLD THE START POSITION OF THE
C SUBSET JUST OBTAINED AND ONE TO STORE THE END POSITION OF THE SUBSET.
C THIS PROGRAM IMPLEMENTS FIG. 7.1 OF CHAPTER 7.

INTEGER L,U
INTEGER*2 ITIME
DIMENSION A(100), IOSTAR(100), IOFIN(100), ITIME(100)
$SHARED A, IOTPUT, IOSTAR, IOFIN, M, ITIME
$USEPAR
READ(6,600) L,U,M, IPRINT
WRITE(6,700) L,U,M
FORMAT(4(I4,1X))
700 FORMAT(1L=14.1X,'U='14.1X,'M='14)
C GENERATE THE INPUT DATA BY THE LIBRARY RANDOM GENERATOR RANGEMANIP
CALL RANGEMANIP
CALL RANGEMANIP RANSETR RANSEP IN THE INTERVAL [0, 1].
DO 4 I=L,U
4 READ(6,900) A(I)
IF(IPRINT.LT.0) WRITE(6,900) A(I)
FORMAT(IX.5(EL:':'-.~,::X))
INITIALISE THE QUEUE.
ISTAR(1)=L
IFIN(1)=U
IOTPUT=1
M=1
L=1
C START THE ALGORITHM TIMING.
DOALL 2
CALL TIMEST
C GET THE START AND END POINTS FOR EACH SUBSET OBTAINED FROM THE QUEUE.
C START PARTITIONING NOW ON THE CURRENT SUBSET.
CALL PART1(A,IS,IE,IN)
IF(IS-IS-IE).GT.IN) GOTO 61
JMIN=IS-IE
JMAX=IE-IN
IF(JMAX.GE.M.AND.JMIN.GE.M) GOTO 312
IF(JMAX.GE.M.AND.JMIN.LT.M) GOTO 119
IF(JMAX.LT.M.AND.JMIN.GE.M) GOTO 312
IF(JMAX.EQ.IS) GOTO 211
CALL QUEUE(IS-IE)
CALL LINSTEM(A,JIS,JIE)
GOTO 50
119 CALL QUEUE(IN)-IE)
CALL LINSTEM(A,IS-IN)
GOTO 50
312 IF(JMAX.EQ.IN-IS) GOTO 112
IF(JMAX.EQ.M.AND.JMIN.GE.M) GOTO 119
IF(JMAX.LT.M.AND.JMIN.LT.M) GOTO 312
IF(JMAX.EQ.IN-IS) GOTO 520
C IF THE SIZE OF ANY SUBSET IS LESS THAN M DO THE INSERTION SORT. OTHERWISE
C PUT ITS START AND END POINTERS IN THE QUEUE FOR FURTHER PARTITIONING.
CALL LINSTEM(A,IS-IE)
C IF THE PROGRAM IMPLEMENTS FIG. 7.1 OF CHAPTER 7.

CALL LINSTEM(A,IS-IN)
GO TO 50
320 CALL LINSTEM(A,IN1-IE)
GOTO 50
71 112 IF(JMAX.EQ.IN-IS) GOTO 117
CALL QUEUE(IS-IN)
CALL QUEUE(IN1-IE)
GOTO 50
75 117 CALL QUEUE(IN1-IE)
CALL QUEUE(IS-IN)
GOTO 50
78 119 IF(JMAX.EQ.IN-IS) GOTO 211
CALL QUEUE(IN1-IE)
CALL LINSTEM(A,IS-IN)
"
GOTO 50
CALL QUEUE(IS,IN-1)
CALL LINSTA(A,IN+1,IE)
GOTO 50
IF(JMAX.ED.IN-IS) GOTO 333
CALL QUEUE(IS,IN-1)
CALL LINSTA(A,IN+1,IE)
GOTO 50
CALL QUEUE(IS+1,IE)
CALL LINSTA(A,IS,IN-1)
GOTO 50
$\text{PARENT}$
C POP THE SUBSETS FROM THE QUEUE AND PERFORM PARTITIONING IN PARALLEL.
IGS=IF+1
IFG=INPUT-1
IF(IGS.LE.IOF) GOTO 10
C IF NO MORE SUBSETS IN THE QUEUE THE ALGORITHM IS COMPLETE END TIMING.
$\text{DOALL}$
CALL TIMOUT(1TIME)
WRITE(6,89)ITIME
FORMAT(1X,8(I6.1X))
IF(IPRINT.GT.1) GOTO 150
WRITE(6,90)IAII).I~L.U)
STOP
C
C CRITICAL SECTION TO UPDATE THE QUEUE THAT HOLDS THE SUBSETS.
SUBROUTINE QUEUE(IK1,K2)
DIMENSION A(2100),IOSTAR(100),IQFIN(100)
$\text{SHARED}$ A,INPUT,ISTAR,IFIN
$\text{REGION}$ QUEUE
$\text{ENTER}$ QUEUE
IOFIN(IINPUT)=IK1
IQSTAR(IINPUT)=IK2
INPUT=INPUT+1
$\text{EXIT}$ QUEUE
RETURN
END
C
C SUBROUTINE TO PARTITION THE SET INTO SMALLER SUBSETS USING THE MEDIAN STRATEGY AS THE PARTITIONING ELEMENT.
SUBROUTINE PARTIA(IK1,K2,K)
DIMENSION A(2100)
AI=A((IK1+K2)/2)
AK1+1)=AI
IF(AIK1+1).GT.AI(2) GOTO 25
IF(AIK1).GT.AI(2) GOTO 47
OOTO 71
AI=AI+1
A(K1+1)=AI
A(K2)=AI
IF(AK1+1).GT.AI(2) GOTO 25
IF(AK1).GT.AI(2) GOTO 47
GOTO 71
AK1=AI
A(K2)=AI
AK1+1)=AI
A(K1)=AI
A(K2)=AI
AK1+1)=AI
AK1+1)=AI
AK1=AI
J=K2
V=AIK1)
I=1
J=1
I=I+1
IF(AI).LT.V.AND.I.LT.J) GOTO 21
J=J-1
IF(AJ).GT.V.AND.J.GT.I) GOTO 23
C THE PARTITIONING IS COMPLETE IF I&J ARE CROSSED.
IF(I.JE.J) GOTO 120
AI=AI
A(I)=A(J)
A(J)=AI
GOTO 21
AI=A(K1)
A(K1)=A(J)
A(J)=AI
K=J
RETURN
SUBROUTINE LINST1(A,N1,N2)
DIMENSION A(2100)
MM=N1-1
MK=N1+1
DO 21 J=MK,N2
R=A(J)
I=J-1
21 IF(R.LT.A(I)) GO TO 22
A(I+1)=R
GOTO 21
A(I+1)=A(I)
I=I-1
22 IF(I.LT.MM) GO TO 24
A(I+1)=R
24 CONTINUE
RETURN
I

PROGRAM A.11

1 2

C THIS PROGRAM IS TO IMPLEMENT THE DPS ALGORITHM OF PROB 7.2 IN THREE
3 4
C PARTS: FIRST, THE MIN AND MAX ELEMENTS ARE FOUND BY ONE PROCESSOR.
5 6
C SECOND PART DISTRIBUTES THE ELEMENTS INTO (U/(2*NPATHS)) BUCKETS AND
7 9
C THIS IS DONE IN PARALLEL. IN THE THIRD PART THE BUCKETS ARE SORTED BY
10 12
C QUICKSORT (THE MEDIAN OF THREE ELEMENTS VERSION). THE SUBSETS ARE
13 15
C PARTITIONED INTO SMALLER ONES UNTIL THERE IS NO SUBSET OF SIZE GREATER
14 17
C THAN (M) TO BE PARTITIONED. THIS PART IS ALSO DONE IN PARALLEL.
15 18

INTEGER L,U
16 19
DIMENSION A(2050),LINK(2050),LH(135),IQS(IOO),IQF(IOO),B(2050),
20 21
IC(2050),IT(135),ITIME(IOO)
22 23
C DEFINE THE SHARED DATA.
24 25
$SHARED A,LH,LINK,IT,C,M,AMIN,AMAX,NDV,NPATHS,CONST,U,ITIME
26 27
~USEPAR
28 29
C READ THE DATA SIZE U, THE CONSTANT TO DECIDE IF TO USE THE QUICKSORT
30 31
C OR THE LINEAR INSERTION SORT, AND THE NUMBER OF PATHS TO GENERATE.
32 33
READ(S,600) U,M,NPATHS,IPRINT
34 35
NP=UINPATHS
36 37
NDV IS THE NUMBER OF BUCKETS.
38 39
NDV=NP/2
40 41
WRITE(6,700) U,M,NPATHS,NDV
42 43
FORMAT(5(I4,I4)
44 45
FORMAT(/'U=',I4,'M=',I4,'NPATHS=',I4,'NDV=',I4)
46 47
THE RANDOM
48 49
CALL RANSET(1)
50 51
DO 19 I=1,U
52 53
A(I)=RANF(DUMMY)
54 55
IF(IPRINT.LT.1)WRITE(6,90)(A(I),I=1,U)
56 57
FORMAT(lX,S(E12.5,2X)
58 59
do 21 I=1,NPV
60 61
LH(I)=O
62 63
AMAX-AMIN
64 65
C START TIMING THE PROCEDURE OF FINDING THE MIN-MAX
66 67
$DOALL 57
68 69
CALL TIMEST
70 71
$PAREND
72 73
C FIND THE MIN-MAX ELEMENTS BY ONLY ONE PROCESSOR.
74 75
DO 19 I=2,U
76 77
IF(A(I).LT.AMIN) GOTO 61
78 79
IF(A(I).LE.AMAX) GOTO 19
80 81
AMAX=A(I)
82 83
GOTO 19
84 85
61 AMIN=A(I)
86 87
CONTINUE
88 89
C CALCULATE THE CONSTANT TO BE USED IN THE DISTRIBUTION.
90 91
CONST=(NDV-0.001)/(AMAX-AMIN)
92 93
C END TIMING OF MIN-MAX PROCEDURE.
94 95
$DOALL 63
96 97
CALL TIMOUT(ITIME)
98 99
$PAREND
100 101
C DISTRIBUTING THE ELEMENTS INTO NDV BUCKETS WHERE P PROCESSORS CAN
102 103
C BE USED. EACH ONE OF THE NDV PATHS HAS A SUBSET OF SIZE 2*NPATHS.
104 105
$DOPAR IP=1,NDV
106 107
IS=2*NPATHS*(IP-1)+1
108 109
IE=2*NPATHS*IP
110 111
IF(IP.EQ.NDV) IE=U
112 113
AMAX=AMIN
114 115
$DOALL 25
116 117
CALL TIMOUT(ITIME)
118 119
$PAREND
120 121
C A CRITICAL SECTION TO UPDATE THE LISTS LINK,LH WHICH ARE USED TO STORE
122 123
C THE POINTERS OF THE ELEMENTS BEING DISTRIBUTED AND THE HEAD OF THE
124 125
C BUCKET RESPECTIVELY, SINCE THEY ARE IN THE SHARED MEMORY.
126 127
CALL UPDATE(L,J)
128 129
CONTINUE
130 131
$PAREND
132 133
C END DISTRIBUTION TIMING.
134 135
$DOALL 3
136 137
CALL TIMOUT(ITIME)
138 139
$PAREND
140 141
142 143
WRITE(6,98)ITIME
144 145
FORMAT(10X/'TIME FOR DISTRIBUTION ONLY'/8(I6,I4)
146 147
C START TIMING FOR THE SORTING PART
CALL TIMES
$DOALL 4
$PAREND
C START SORTING THE NDV BUCKETS BY THE QUICKSORT IF THEY ARE >M ELSE USE THE LINEAR INSERTION SORT. THIS PART IS CARRIED OUT IN PARALLEL.
$DOPAR 50 IF=1,NDV
IOPUT=1
IGT=1
IH=LH(IP)
KTT=1
K4=KTT
K3=KTT
IF(IH.EQ.0) GOTO 93
C COLLECT THE ELEMENTS OF THE BUCKET IP AND PUT THEM IN A LOCAL VECTOR B.
B(KTT)=A(IH)
KTT=KTT+1
IH=LINK(IH)
IH=IHI
GOTO 91
ITI=KTT-1
ITT=ITI
IF(ITT.NE.0) GOTO 269
IT(IP)=0
GOTO 50
C START SORTING THE VECTOR B
IF(ITT-KT+1.GT.M) GOTO 10
CALL LINSTIB(KT,ITT)
GOTO 259
CALL PARTIB(KT,ITT,IN)
JMI-IN-KT
JM2=ITT-IN
IFIJM2.GE.JMI) GOTO 112
GOTO 103
IFIJM2.LE.JMI) GOTO 217
CALL LINSTIB(IN+I,ITT)
CALL LINSTIB(IN-I,ITT)
GOTO 55
IFIJMI.GT.M.AND.JM2.LE.M) GOTO 216
CALL LINSTIB(IN+I,ITT)
CALL LINSTIB(IN-I,ITT)
GOTO 55
IFIJMI.LE.M.AND.JM2.GT.M) GOTO 319
CALL LINSTIB(IN+I,ITT)
CALL LINSTIB(IN-I,ITT)
GOTO 55
CALL QUUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
CALL QUEUE(IOPUT,IGS,IGF,KT.IN+1,ITT)
CALL QUEUE(IOPUT,IGS,IGF,KT.IN-1,ITT)
Puting IN THE QUEUE THE SUBSETS OBTAINED FROM THE QUICKSORT'S PARTITIONING.
C AFTER THE BUCKET IS SORTED PUT THE ELEMENTS INTO THE SHARED VECTOR C.

DO 123 IS=K,ITI
C(JY)=B(IS)
JY=JY+1
123 CONTINUE
IT(LB+1)=JY
DO 192 IV=1,10T
192 IGS(IV)=0
190 50 $PEND
171 C END SORTING TIMING.
$DOALL 9
CALL TIMOUT(ITIME)
PEND(6,69)ITIME
FORMAT(80SORTING TIMING/8(16,1X))
IF(IPRINT.OF.1) GOTO 150
WRITE(6,90)ICCI),I=1,U)
$STOP
$END

C SUBROUTINE TO UPDATE THE QUEUE THAT HOLDS THE SUBSETS.

C SUBROUTINE TO PARTITION THE SET INTO SMALLER SUBSETS USING THE MEDIAN ELEMENT.

SUBROUTINE PART1(Y,K1,K2,K)
DIMENSION Y(2050)
AI=Y((K1+K2)/2)
Y(K1+1)=AI
IF(Y(K1).LT.Y(K2))GOTO 25
IF(Y(K1).GT.Y(K2))GOTO 37
GOTO 71
25 AI=Y(K1+1)
20 Y(K1+1)=Y(K2)
20 Y(K2)=AI
20 GOTO 71
20 Y(K1)=Y(K2)
20 Y(K2)=AI
20 GOTO 71
20 Y(K1)=Y(K2)
20 Y(K2)=AI
20 GOTO 71
20 Y(K1)=Y(K2)
210 AI=Y(K1+1)
210 Y(K1+1)=Y(K1)
20 Y(K1)=AI
210 J=K2
210 V=Y(K1)
216 I=I+1
216 J=J+1
219 IF(Y(I).LT.V.AND.I.LT.J) GOTO 21
220 J=J-1
221 IF(Y(J).GT.V.AND.J.GT.I) GOTO 23
222 C THE PARTITION IS COMPLETE IF I,J ARE CROSSED.
222 IF(I.GE.J) GOTO 120
222 AI=Y(I)
222 Y(I)=Y(J)
222 Y(J)=AI
222 GOTO 21
222 AI=Y(K1)
222 Y(K1)=Y(J)
222 Y(J)=AI
222 K=J
232 RETURN
233 END

C THE MEDIAN

C THIS SUBROUTINE TO SORT THE SUBSETS OF SIZE LESS THAN M BY USING THE LINEAR INSERTION SORT.

SUBROUTINE LINST1(Y,N1,N2)
DIMENSION Y(2050)
MM=N1-1
241 MK=NI+1
242 DO 21 J=MK,N2
243 R=Y(J)
244 I=J-1
245 24 IF(R.LT.Y(I)) GOTO 22
246 Y(I+1)=R
247 GOTO 21
248 22 Y(I+1)=Y(I)
249 I=I-1
250 IF(I.GT.MM) GOTO 24
251 Y(I+1)=R
252 21 CONTINUE
253 RETURN
254 END

C CRITICAL SECTION TO UPDATE THE LIST(LINK) THAT HOLDS THE POINTERS OF THE
C ELEMENTS AND (LH) THAT HOLDS THE HEAD POINTER OF THE LIST(LINK).
255 SUBROUTINE UPDATLK1,K2)
256 DIMENSION A(2050),LH(135),LINK(2050)
257 $SHARED A,LH,LINK
258 $REGION UPDATL
259 $ENTER UPDATL
260 LINK(K2)=LINK(K1)
261 LH(K1)=K2
262 $EXIT UPDATL
263 RETURN
264 END
THIS PROGRAM IMPLEMENTS THE DPS ALGORITHM OF FROG 7.5. THE ELEMENTS ARE PARTITIONED INTO NPATHS 'BUCKETS', WHERE EACH BUCKET IS ASSIGNED TO A PROCESSOR. EACH PROCESSOR THEN SortS ITS BUCKET BY USING THE QUICKSORT (THE MEDIAN OF THREE ELEMENTS VERSION) IF THE BUCKET IS LARGE, OTHERWISE THE LINEAR INSERTION SORT IS USED. AT THE END OF THIS PROCEDURE THE BUCKETS WILL BE SORTED WITHIN THEMSELVES BUT NOT AMONG EACH OTHER.

THE 2-WAY MERGE ALGORITHM IS USED TO MERGE THE SUBSETS TO OBTAIN THE FINAL SORTED SET.

INTEGER L,U
INTEGER*2 ITIME

DIMENSION A(2050),LINK(2050),LH(129),IOS(100),IOP(100),B(2050),
L(2050),ITIME(100)

DEFINE THE SHARED DATA.

REGION UPDATE

READ(5,600) U,M,NPATHS,LOOP,PRINT

U=THE DATA SIZE. M=CONSTANT TO DECIDE IF TO SORT BY QUICKSORT OR LINEAR INSERTION SORT, NPATHS=THE NUMBER OF PATHS GENERATED, LOOP=LOGCNPATHS).

GENERATE THE RANDOM NUMBERS BY THE LIBRARY ROUTINES RANSET AND RANF IN THE INTERVAL[0,1].

CALL RANSET(00 105 1=I.U)
ACI)=RANF(DUMMY)
IF(CPRINT.LT.1)WRITE(o.90) (ACI).I=1.U)
FORMAT(1X.5(E12.5.2X)

OBTAIN THE SIZE OF EACH BUCKET.
NELM=U/NPATHS

START SORTING TIMING.

THIS LOOP TO BE PERFORMED IN PARALLEL. EACH PROCESSOR SORTS ITS SUBSET BY THE DPS ALGORITHM.

DOALL IS=1,NPATHS

IF(IP.EQ.NPATHS) IE=U

FIND THE MIN&MAX ELEMENTS.
DO 67 43-1,NPATHS

LL=IS+l
DO 19 t=LL, lE
IF(CAI).LT.AMIN) GOTO 61
IF(CAI).LE.AMAX) GOTO 19
AMAX=ACAI)
GOTO 19
AMIN=ACAI)
CONTINUE

CONST=CNPATHS-0.001)/CAMAX-AMIN)
DISTRIBUTE THE ELEMENTS INTO NPATHS BUCKETS. THE LISTS (LINK)&(LH) ARE RESPECTIVELY TO STORE THE POINTERS CORRESPONDING TO THE ELEMENTS BEING DISTRIBUTED INTO A BUCKET. AND TO STORE THE HEAD POINTER FOR EACH BUCKET.

DO 67 J=1,NPATHS

IF(A(I).LT.AMIN) GOTO 61
IF(A(I).LE.AMAX) GOTO 19

AMAX=A(I)
GOTO 19
AMIN=A(I)
CONTINUE

DISTRIBUTE THE ELEMENTS INTO NPATHS BUCKETS. THE LISTS (LINK)&(LH) ARE RESPECTIVELY TO STORE THE HEAD POINTER FOR EACH BUCKET.

DO 67 J=1,NPATHS

IF(IH.EQ.0) GOTO 763
SI C COPY THEM INTO A LOCAL VECTOR B.
B(KTT)=A(IH)
IFT(IH).EQ.0 GOTO 93
IH=IH
GOTO 191

IF THE SIZE OF THE BUCKET>M THEN USE QUICKSORT ELSE INSERTION SORT
IF(ITI-KT+1.GT.MS) GOTO 10
CALL LINST1(B.KT,ITI)
GOTO 259

CALL PART1(B,KT,ITI,IN)
THE PARTITIONING PROCEDURE OF THE QUICKSORT IS USED AGAIN IF THE SUBSET
C IS GREATER THAN M ELSE THE LINEAR INSERTION SORT IS USED.
JM1=IN-KT
JM2=ITI-IN
IF(JM2.LT.JM1) GOTO 113
IF(JM2.GT.MS.AND.JM1.GT.MS) GOTO 117
IF(JM2.GT.MS.AND.JM1.LE.MS) GOTO 217
IF(JM2.LE.MS.AND.JM1.GT.MS) GOTO 317
CALL LINST1(B.IN+1,ITI)
CALL LINST1(B.KT,IN-1)
ADD TO THE QUEUE THE SUBSETS OBTAINED FROM PARTITIONING
CALL QUEUECPUT(IQS,IQF,KT,IN-1)
CALL LINST1(B.KT,IN-1)
CALL QUEUECPUT(IQS,IQF,KT,IN-1)
ADD TO THE QUEUE SUBSET AFTER ANOTHER UNTIL THE QUEUE IS EMPTY.
IF(IQS(IQF).EQ.O.AND.IQF(IQF).EQ.O) GOTO 259
KT=IQT
IT1=QF(IQT)
IQT=IQT+1
CONTINUE

A CRITICAL SECTION TO PUT THE ELEMENTS INTO A SHARED VECTOR C
ENTER UPDATED
DO IG=10,10+1
C(IG)=B(14)
CONTINUE
EXIT UPDATED
DOALL 7
END SORT TIMING NOW.

WRITE THE TIMING ARRAY AND THE OUTPUT VECTOR OF THE PARTIAL RESULTS.
WRITE(16,555)ITIME
FORMAT('THE TIME FOR SORTING'/S(16.1X)
IF(IPRINT.LT.1) WRITE(6,90)
IF(LOGP.EQ.O) GOTO 110
WRITE(6,90)
CALL TIMOUT(ITIME)

WRITE THE TIMINO ARRAY AND THE OUTPUT VECTOR OF THE PARTIAL RESULTS. WRITE(6,555)ITIME
FORMAT('THE TIME FOR SORTING'/S(16.1X)
IF(IPRINT.LT.1) WRITE(6,90)
IF(LOGP.EQ.O) GOTO 110
WRITE(6,90)
CALL TIMOUT(ITIME)

START TIMING THE MERGE PROCEDURE.
DOALL 5,8
CALL TIMES
162 5 5 *PAREND
163 C START MERGING THE SORTED SUBSETS OBTAINED FROM THE ABOVE DOPAR LOOP BY
164 C USING THE 2-WAY MERGE THAT REQUIRES LOOP SEQUENTIAL STEPS.
165 DO 100 KO=1,LOGP
166 C DETERMINE THE NUMBER OF SUBSETS OF EACH STEP
167 NP=NPATHS/2**KO
168 C THIS LOOP IS PERFORMED IN PARALLEL
169 C DOPAR 50 IP=1, NP
170 C CALCULATE THE START-END POSITIONS OF THE TWO SUBSETS BEING MERGED.
171 IPS=2**KG*NELM*(IP-1)+1
172 IPE=2**KG*(IP)+NELM*(2*IP-1)
173 J1=IPS+NELM*2**(KG-1)
174 J2=IPE+NELM*2**(KG-1)
175 L1=IPS
176 L2=J1
177 123 IPS-2**KG*NELM*(IP-1)+1
178 IPE=2**KG*(IP)+NELM*(2*IP-1)
179 J1=IPS+NELM*2**(KG-1)
180 L2=J1
181 124 C COMPARE THE ELEMENTS OF THE FIRST SUBSET WITH THAT OF THE SECOND UNTIL
182 C ALL ELEMENTS ARE ARRANGED IN THEIR FINAL POSITIONS. THE PARTIAL RESULTS
183 C ARE KEPT IN THE LOCAL VECTOR B.
184 DO 12=12+1
185 IF(C(L1).LT.C(L2)) GOTO 70
186 B(12)=C(L2)
187 L2=L2+1
188 IF(L2.LE.J2) GOTO 22
189 GOTO 37
190 37 DO 12=12+1
191 B(12)=C(12)
192 12 CONTINUE
193 22 C PUT THE ELEMENTS BACK INTO THE SHARED VECTOR C
194 43 DO 14=LI,J2
195 14 CONTINUE
196 C AFTER ALL PROCESSORS HAVE FINISHED THEIR MERGE IN THE PREVIOUS STEP DO
197 C THE NEXT STEPS.
198 DO 100 CONTINUE
199 110 CALL TIMOUT(ITIME)
200 6 SPAREND
201 WRITE(6,198)ITIME
202 IF(IPRINT.GT.I) GOTO 150
203 WRITE(6,90)(C(I)),I=1,IPRINT
204 90 STOP
205 C SUBROUTINE TO UPDATE THE QUEUE THAT HOLDS THE SUBSETS
206 219 C SUBROUTINE TO UPDATE THE QUEUE THAT HOLDS THE SUBSETS
207 CALL TIMOUT(ITIME)
208 6 *PAREND
209 WRITE(6,198)ITIME
210 IF(IPRINT.GT.I) GOTO 150
211 WRITE(6,90)(C(I)),I=1,IPRINT
212 END
213 C SUBROUTINE TO PARTITION THE SET INTO SMALLER SUBSETS USING THE MEDIAN
214 C OF THREE STRATEGY AS THE PARTITIONING ELEMENT.
215 SUBROUTINE PARTI(B,KI,K2,K)
216 DIMENSION 8(2050)
217 A1=B(KI+K2)/2
218 B(KI+1)=A1
219 IF(B(KI+1).GT.B(K2)) GOTO 25
220 IF(B(KI+1).LT.B(K2)) GOTO 25
221 DO 100 CONTINUE
222 C SUBROUTINE TO PARTITION THE SET INTO SMALLER SUBSETS USING THE MEDIAN
223 C OF THREE STRATEGY AS THE PARTITIONING ELEMENT.
224 SUBROUTINE PARTI(B,KI,K2,K)
225 DIMENSION B(2050)
226 A1=B(KI+K2)/2
227 B(KI+1)=A1
228 IF(B(KI+1).GT.B(K2)) GOTO 25
229 IF(B(KI+1).LT.B(K2)) GOTO 25
230 DO 100 CONTINUE
231 SUBROUTINE B(KI,K2,K)
232 219 C SUBROUTINE TO UPDATE THE QUEUE THAT HOLDS THE SUBSETS
233 220 C SUBROUTINE TO UPDATE THE QUEUE THAT HOLDS THE SUBSETS
234 221 DIMENSION 10S(100),IQS(IOPUT),IOF(IQF)
235 IOF(IOPUT)=K1
236 IQS(IOPUT)=K2
237 IOPUT=IOPUT+1
238 END
239 C SUBROUTINE TO UPDATE THE QUEUE THAT HOLDS THE SUBSETS
240 C SUBROUTINE TO UPDATE THE QUEUE THAT HOLDS THE SUBSETS
241 240 B(KI+1)=B(K2)
B(K2)=A1
GOTO 72
A1=B(K1)
B(K1)=B(K2)
B(K2)=A1
GOTO 73
A1=B(K1+1)
B(K1+1)=B(K1)
B(K1)=A1
I=K1+1
J=K2
V=B(K1)
I=1
J=J
IF(B(I).LT.V.AND.I.LT.J1) GOTO 21
IF(B(J).GT.V.AND.J.GT.I1) GOTO 23
C PARTITIONING IS COMPLETE IF I&J ARE CROSSED.
IF(I.GE.J)GOTO 120
A1=B(I)
B(I)=B(J)
B(J)=A1
GOTO 21
A1=B(K1)
B(K1)=B(J)
B(J)=A1
K=J
RETURN
END

C THIS SUBROUTINE TO SORT THE SUBSETS OF SIZE LESS THAN M BY USING THE
LINEAR INSERTION SORT.
DIMENSION B(2050)
MM=N1-1
MK=N1+1
DO 21 J=MK,N2
R=B(J)
I=J-1
IF(R.LT.B(I)) GOTO 22
B(I+1)=R
GOTO 21
B(I+1)=B(I)
I=I-1
IF(I.GT.MM) GOTO 24
B(I+1)=R
CONTINUE
RETURN
END


[Cray 75]: "The CRAY-1 Computer", Preliminary Reference Manual, Cray

Csanky L. [1975]: "Fast Parallel Matrix Inversion Algorithms", 16th Annual
Symposium on Foundations of Computer Science (SWAT), Barkely, 1975,
pp.11-13.

Dennis J.B. and Van Horn E.C. [1966]: "Programming Semantics for Multi-

Dennis J.B. and Misunas, D.P. [1975]: "A Preliminary Architecture for a
Basic Data Flow Processor", Conf. Proc. of the 2nd Annual Symposium

Dijkstra, E.W. [1968]: "Co-operating Sequential Processes", in 'Programming
Languages', ed. by F. Genuys, IBM Paris, France, Academic Press, 1968,
pp.43-112.

Dobosiewicz W. [1978]: "Sorting by Distributive Partitioning", Information

Donnelly, J.D.P. [1971]: "Periodic Chaotic Relaxation", Linear Algebra

Thesis, Computer Studies Department, Loughborough University.

Enslow, P.H. [1974]: "Multiprocessors and Parallel Processing", Ed. by
Enslow, A Wiley-Interscience Publication, John Wiley and Sons, New
York.


Flanders P.M. [1982]: "Languages and Techniques for Parallel Array Processing", Ph.D. Thesis, Queen Mary College, University of London.


