## This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.Iboro.ac.uk/) under the following Creative Commons Licence conditions.

## (c) creative

C O M M O N S D E E D

Attribution-NonCommercial-NoDerivs 2.5

You are free:

- to copy, distribute, display, and perform the work

Under the following conditions:

BY Attribution. You must attribute the work in the manner specified by the author or licensor


Noncommercial. You may not use this work for commercial purposes.

No Derivative Works. You may not alter, transform, or build upon this work

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Leqal Code (the full license).
Disclaimer $\left.{ }^{[ }\right]$

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/


# PARALLEL ALGORITHMS FOR MIMD PARALLEL COMPUTERS 

## BY

Hilal Mohammad Yousif, B.Sc.,M.Sc.

A Doctoral Thesis submitted in partial fulfilment of the award of Doctor of Philosophy of the Loughborough University of Technology

September, 1986.

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

$$
\begin{aligned}
& \text { L.ougheremg imetsity }
\end{aligned}
$$

$$
\begin{aligned}
& \mathrm{D}_{2}:-\mathrm{Feb} .87 \\
& \text { Cis: } \\
& \text { Acc. } 012359 / 02
\end{aligned}
$$

## 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.

## To My Mother,

## The memory of my Father,

My wife, MARIAM,
without whose support and
encouragement this work would not have been possible,

My daughters, RASHA, RANA,

And my son, ALI.

## ACKNOWLEDGEMENTS

I wish to acknowledge my sincere gratitude to my supervisor, Professor D.J. Evans, for his kind advice, invaluable guidance and continued encouragement and assistance during the past three years of study and preparation of this thesis.

I acknowledge also with gratitude the IRAQI Government for their support to complete this thesis.

A sincere gratitude and thanks to Gen. K.A. AL-AZZAWI for his encouragement and support during the course of this thesis.

Acknowledgements are also presented to the following people:

- Mr. R.P. Stallard for his cooperation and help with the parallel system questions and requirements.
- Dr. W.S. Yousif for his valuable discussions and comments in private communications.
- Dr. A.M.S. Rahma for his help and valuable discussion in the early stages of this work.


# Parallel Algorithms for Mimd Parallel Computers 

## ABSTRACT

This thesis mainly covers the design and analysis of asynchronous parallel algorithms that can be run on MIMD (Multiple Instruction Multiple Data) parallel computers, in particular the NEPTUNE system at Loughborough University. Initially the fundamentals of parallel computer architectures are introduced with different parallel architectures being described and compared. The principles of parallel programming and the design of parallel algorithms are also outlined. Also the main characteristics of the 4 processor MIMD NEPTUNE system are presented, and performance indicators, i.e. the speed-up and the efficiency factors are defined for the measurement of parallelism in a given system.

Both numerical and non-numerical algorithms are covered in the thesis. In the numerical solution of partial differential equations, a new parallel 9-point block iterative method is developed. Here, the organization of the blocks is done in such a way that each process contains its own group of 9 points on the network, therefore, they can be run in parallel. The parallel implementation of both 9-point and 4point block iterative methods were programmed using natural and redblack ordering with synchronous and asynchronous approaches. The results obtained for these different implementations were compared and analysed.

Next the parallel version of the A.G.E. (Alternating Group Explicit) method is developed in which the explicit nature of the difference equation is revealed and exploited when applied to derive the solution
of both linear and non-linear 2-point boundary value problems. Two strategies have been used in the implementation of the parallel A.G.E. method using the synchronous and asynchronous approaches. The results from these implementations were compared. Also for comparison reasons the results obtained from the parallel A.G.E. were compared with the corresponding results obtained from the parallel versions of the Jacobi, Gauss-Seidel and S.O.R. methods. Finally, a computational complexity analysis of the parallel A.G.E. algorithms is included.

In the area of non-numeric algorithms, the problems of sorting and searching were studied. The sorting methods which were investigated was the shell and the digit sort methods. With each method different parallel strategies and approaches were used and compared to find the best results which can be obtained on the parallel machine.

In the searching methods, the sequential search algorithm in an unordered table and the binary search algorithms were investigated and implemented in parallel with a presentation of the results. Finally, a complexity analysis of these methods is presented.

The thesis concludes with a chapter summarizing the main results.

## Contents

PAGE
CHAPTER 1: FUNDAMENTALS OF PARALLLEL COMPUTER ARCHITECTURES ..... 1
1.1 Introduction ..... 2
1.2 Towards Parallel Processing ..... 4
1.3 The Structure of Parallel Computers ..... 7
1.3.1 Pipeline Computers ..... 7
1.3.2 Array Processors ..... 17
1.3.3 Multiprocessor Systems ..... 20
1.4 Architectural Classification Schemes ..... 30
1.4.1 Flynn's Classification ..... 30
1.4.2 Shore's Classification ..... 31
1.5 SIMD Machine ..... 36
1.6 MIMD Computers ..... 44
1.7 Loughborough University Parallel Systems ..... 62
1.8 Data-Flow Computers ..... 68
1.9 VLSI Systems ..... 76
CHAPTER 2: PARALLEL PROGRAMMING PRINCIPLES ..... 83
2.1 Introduction ..... 84
2.2 Hierarchies of Parallelism ..... 86
2.2.1 Explicit Parallelism ..... 89
2.2.2 Implicit Parallelism ..... 96
2.3 Parallel Languages for Vector Processing ..... 104
2.4 Array Processing Languages and Programming ..... 109
2.5 MIMD Multiprocessor Programming ..... 116
2.6 Dat́a Flow Languages ..... 120
2.7 Parallelism in High-Level Programming ..... 126 Languages
2.8 Programming the NEPTUNE System ..... 135
CHAPTER 3: DESIGN OF PARALLEL ALGORITHMS ..... 142
3.1 Introduction ..... 143
3.2 Parallel Algorithms for SIMD and PIPELINE ..... 147 Computers
3.3 Parallel Algorithms for MIMD Computers ..... 158
3.4 VLSI and Systolic Algorithms ..... 166
3.5 Parallel Numerical Algorithms ..... 179
3.5.1 Inherent Parallelism ..... 180
3.5.2 Vector Iterative Method ..... 184
3.5.3 Recurrence Relations ..... 186
3.6 Parallel Non-Numerical Algorithms ..... 192
3.7 Performance of Parallel Computer Algorithms ..... 200
CHAPTER 4: PARALLEL IMPLEMENTATIONS OF THE FOUR-POINT AND NINE- POINT EXPLICIT-BLOCK ITERATIVE METHODS ..... 203
4.1 Introduction ..... 204
4.2 Basic Definitions ..... 206
4.3 Basic Methods for Solving a Set of Linear Equations ..... 217
4.3.1 Direct Methods ..... 217
4.3.2 Iterative Methods ..... 223
4.4 Convergence of Point Iterative Methods ..... 228
4.5 Rate of Convergence ..... 230
4.6 The Optimum Accelexation Parameter for the ..... 232 SOR Method
4.7 Finite Difference Approximations to Derivatives ..... 235
4.8 Block Iterative Methods ..... 243
4.9 Parallel Iterative Methods ..... 251
4.10 The Four-Point Explicit Block Iterative ..... 258 Method
4.11 The 9-Point Explicit Block Iterative ..... 263 Method
4.12 Experimental Results of the Block Iterative ..... 278 Methods
4.13 Conclusions ..... 332
CHAPTER 5: THE PARALLEL ALTERNATING GROUP EXPLICIT (A.G.E.) ..... 334
METHOD
5.1 Introduction ..... 335
5.2 Alternating Direction Implicit (A.D.I.) ..... 339 Methods
5.3 The Alternating Group Explicit (A.G.E.) ..... 342
Method
5.4 Experimental Results of the Parallel A.G.E. ..... 348 Method
5.5 Conclusions ..... 412
CHAPTER 6: PARALLEL SORTING AND SEARCHING ALGORITHMS ..... 414
6.1 Introduction ..... 415
6.2 The Sorting Algorithms ..... 420
6.2.1 The Shell Sort Method ..... 420
6.2.2 The Digit Sort Method ..... 441
6.3 The Searching Algorithms ..... 466
6.3.1 Sequential Search (Unordered Table) ..... 467
6.3.2 Binary Search ..... 479
6.4 Conclusions ..... 488
CHAPTER 7: SUMMARY AND CONCLUSIONS ..... 492
APPENDIX: SELECTED COMPUTER PROGRAMS ..... 502
REFERENCES ..... 562

Chapter 1

### 1.1 INTRODUCTION

The greatest possible speed, throughput, performance, flexibility and a high level of availability and reliability is the requirement to many scientific and engineering applications, many of which need to be solved in real time. Since the speed and reliability of conventional computers is limited, the satisfaction of these requirements can only be achieved by a high-performance computer system. Achieving high performance depends not only on using faster and more reliable hardware devices but also on different computer architectures and processing techniques, so parallel computer systems need to be developed.

In the early days of computers, vacuum tubes were used and hardware components were expensive, heat generating and slow. CPU structure was bit-serial, and arithmetic was done on a bit-by-bit fixed point basis. Transistors were invented in 1948 and the first transistorised digital computer was built in 1954. Printed circuits appeared and magnetic core memory was developed with the result that many computer systems now use it. In 1959, many improvements to computer architecture were carried out. For example, Sperry Rand built a computer system with an independent I/O processor which operated in parallel with one or two processing units. Between the early 1960's and mid-1970's, small-scale integrated (SSI) and medium-scale integrated (MSI) circuits were used as basic blocks in computer systems. Core memory was still used in many computer systems, like the CDC-6600. Then solid-state memories replaced the core memories and many fast computers like CDC-7600 were built. In the early seventies, many high-performance computers were developed, i.e., such computer systems were the IBM 360/91, Illiac IV, TI-ASC, Cyber-175, STAR-100 and C.mmp, and several vector processors were also developed.

The present and near future computer designs emphasize the use of large-scale integrated (LSI) circuits and very-large-scale (VLSI) chips, for both logic and memory sections. Second generation vector machines have appeared like the Cray-1 (1976) and Cyber-205 (1982) and high-speed mainframe and super computers appear in multiprocessor system form like the Univac 1100/80, (1970), the IBM 3081 (1980), and the Burroughs B-7800 (1978). In addition, a high degree of pipelining and multiprocessing is emphasized in commercial super-computers.

From the above history of computer developments, the size of computers has decreased and the speed of operation has increased rapidly. Two methods can be used to achieve the highest possible speed and throughput. These methods are: a) by exploiting the technological possibilities in the design of the computer components and b) by a suitable modification to the structure and organization of the computer. Since the increase in speed and density is not unlimited, for it has ultimate physical limits then the other method of increasing speed, improving the organization architectures of computers, leads to the design of parallel architectures of computers.

### 1.2 TOWARDS PARALLEL PROCESSING

From a user's point of view, a high degree of parallelism is needed as the computer applications become more and more sophisticated. Computer usage started with data processing (i.e. a collection of raw data such as . . numbers, characters, symbols,...) and as computer science theory has advanced, many users have shifted to information processing (i.e., data objects related by some syntactic relations). A high degree of parallelism has been found at these two levels of computer applications. In recent years, a knowledge based application system (where knowledge consists of information items plus some semantic meanings) has been established where a greater degree of parallelism is needed and it is more exploitable in this type of application than that in the data processing and information processing applications fields.

From an operating system point of view, four operating modes have been in usage for successive computer systems. These modes are: batch processing, multiprogramning, time sharing, and multiprocessing. Each mode has been an improvement to its predecessor. The degree of parallelism has increased from one mode to another sharply. Hwang and Briggs [1984] formally defined parallel processing as follows: Parallel processing is an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process. Concurrency implies parallelism, simultaneity, and pipelining. Parallel events may occur in multiple resources during the same time interval; simultaneous events may occur at the same time instant; and pipelined events may occur in an overlapped time span.

Also they show that, parallelism has been applied at several distinct levels which might be classified as:

1. Job or program level,

This is the highest level which can be conducted among multiple jobs or programs through multiprogramming, time sharing and multiprocessing.
2. Task or procedure level,

This is the next highest level of parallel processing and is conducted among procedure or tasks (program segments) within the same program.
3. Interinstruction level,

This level is to exploit concurrency among multiple instructions.
4. Intrainstruction level,

This is the lowest level to obtain faster and concurrent operations within each instruction.

These levels are implemented either by hardware or software means. Hardware roles increase from high to low levels, while software implementations increase from low to high levels.

To achieve parallel processing requires the development of more capable and cost-effective computer systems. Since the conventional uniprocessor system have their limit in achieving high performance, the general architectural trend is being shifted away from conventional uniprocessor systems to multiprocessor systems or to an array of processing elements controlled by one uniprocessor. In all cases, a high degree of pipelining is being incorporated into the various system levels.

Hockney and Jesshop [1981] summarised the principal ways of introducing parallelism into the architecture of computers as:

1. The application of pipelining techniques to improve the performance of arithmetic or control units.
2. Providing several functional units, such as logic, addition, and multiplication, where these functions operate in parallel on different data.
3. Provide an array or vector of processing elements performing the instruction simultaneously, but on different data, where the data is stored in the processing elements' private memories.
4. Providing several processors each of them being a complete computer.

### 1.3 THE' STRUCTURE OF PARALLEL COMPUTERS

Parallel computers are simply those systems carrying out several operations at the same time, or those systems which are carrying out parallel processing. A parallel computer structure will be characterised as:

1. Pipeline computers
2. Array processors
3. Multiprocessor systems.
$\therefore$ UThe pipelining principle implies the segmentation or partition of a computitiogel process into $N$ subprocesses which can be executed independentiy by ditstinct units or modules. An array processor uses $\because \%$
multiple prootssing units in a synchronized manner. A multiprocessor system uses a set of asynchronized processors with shared resources. Some gcinputers belong to more than one of the above characterizations. New concepts such as data flow and VLSI approaches will also be introduced in later sections.

### 1.3.1 Pipeline Computers

The principle of pipelining has emerged as a major architectural attribute of most present day computer systems. The concept of pipeline processing in a computer is similar to assembly lines in an industrial plant. To achieve pipelining, one must subdivide the input task (process) into a sequence of subtasks, each of which can be executed by a specialized hardware stage that operates concurrently with other stages in the pipeline.
pipelined machines such as the CDC STAR-100 (Hintz, R.G. and

Tate, D.P. 1972), CRAY-1 (Cray 1975) and the Texas Instrument ASC (Watson, 1972) have different pipeline processing capabilities, which varies from pipelined special purpose functional units to internally pipelined instruction and arithmetic units.

A pipeline processor consists of a sequence of processing circuits, called segments or stages through which a data stream passes (Figure 1.1). Each segment does some partial processing, on the data and a final result is obtained after the data has passed through all the segments of the pipeline. Parallel processing is achieved by having distinct operand sets or processes in several segments at the same time.


Segment m

Output Register

R: Register
C: Processing Circuit

A typical pipeline computer is shown in Figure 1.2. This model shows both scalar arithmetic pipelines and vector arithmetic pipelines. The instruction processing unit is itself pipelined with three stages as shown.


O: Operand fetch
K: Control signal
FIGURE 1.2: Functional structure of a modern pipeline computer with scalar and vector capabilities

As an example, consider the process of executing an instruction. Normally it involves the operations of fetching the instruction (F), decode the instruction (D), fetch the operand ( 0 ), and finally its execution (E). In a non-pipelined computer, the above steps must be completed before the next instruction can be issued as shown in Figure 1.3.


FIGURE 1.3: Non-pipelined processor

In a pipelined computer, successive instructions are executed in an overlapped fashion. The four pipeline stages $F, D, O$ and $E$ which are connected serially are shown in Figure l.4. After constant time intervals, the output of one stage is shifted to the next. A new instruction is fetched (F) in every time cycle, and stage (E) produces an output every time cycle because the time to execute an instruction consists of multiple pipeline cycles. For the nonpipelined (nonoverlapped) computer, it takes four pipeline cycles to complete one instruction. While in pipelined computers, once the pipeline is filled up, an output result is produced on each cycle. So by overlapping, the execution time the instruction processing will be faster by a factor of four (in our example, instruction processes in four stages) over nonpipelined execution. The two space-time diagram (Figure 1.5 and Figure 1.6) shows the difference between overlapped instruction execution and sequentially nonoverlapped execution.


FIGURE 1.4: A pipelined processor to execute an instruction


FIGURE 1.5: Space-time diagram for a pipelined processor


FIGURE 1.6: Space-time diagram for a nonpipelined processor

### 1.3.1.1 Classification of Pipeline Processors

Handler (1977) classified the pipeline processors into three classes according to the levels of processing i.e.:

1. Arithmetic pipelining

Most arithmetic functions are easily implemented by pipelining. Arithmetic pipelines have been constructed for performing either a single arithmetic function or performing all the four basic operations on both fixed-point and floating-point numbers. As examples we have the four-stage pipes used in STAR-100, the eight-stage pipe used in TI-ASC, the up to 14 pipeline stages used in the CRAY-1, and the up to 26 stages per pipe in the Cyber-205.

## 2. Instruction pipelining

The purpose of an instruction pipeline is to overlap the execution of the current instruction with the subsequent instruction stages. Almost all high-performance computers are now equipped with instructionexecution pipelines.

## 3. Processor pipelining

In this type of pipelining, a cascade of processors, each with a specific task,process the same data stream. The result from the first processor is passed on to the second processor, and the result from the second processor is passed to the third processor etc.

Ramamoorthy and Li (1977) have proposed the following classification scheme:

1. Unifunction vs. Multifunction Pipelines

A pipelined module that only serves a single dedicated function such as the floating-point adder, is called unifunctional. The CRAY-1
has 12 unifunctional pipeline units. A multifunctional pipe may perform different functions, either at a different time or at the same time, by interconnecting different subsets of stages in the pipeline. The TI-ASC has four multifunction pipeline processors.

## 2. Static vs. Dynamic Pipelines

A static pipeline may assume only one functional configuration at a time. It can be either unifunctional or multifunctional. On the other hand, a dynamic pipeline processor permits several functional configurations to exist simultaneously and must be maltifunctional.

## 3. Scalar vs. Vector Pipelines

A scalar pipeline processes a sequence of scalar operands under the control of a DO loop. A typical example of a machine equipped with scalar pipelines is the IBM 360/91. Vector pipelines are specially designed to handle vector instructions over vector operands. Examples of vector pipeline machines are IT-ASC, CDC STAR-100, CDC Cyber-205 and CRAY-1.

Block diagrams for the pipeline machines CDC STAR-100, CRAY-1 and IT-ASC are shown in Figures $1.7,1.8$ and 1.9 respectively.


Direct access
channel
FIGURE 1.7: A system architecture of CDC STAR-100


FIGURE 1.8: Block diagram of the CRAY-1 computer


FIGURE 1.9: Block diagram of the Texas Instrument ASC

It can be shown that a pipeline processor with n-stages could be at most $n$ times faster than a nonpipeline processor. From Figure 1.4, to process \& tasks using a linear pipeline with $n$ stages, it takes

$$
T_{\mathrm{n}}=\mathrm{n}+(\ell-1) \text { clock periods. }
$$

(The clock period of a linear pipeline is equal to the maximum delay in each stage $s_{i}$ plus a time delay between'each stage), where $n$ cycles are used to complete the execution of the first task and ( $\ell-1$ ) cycles are required to complete the remaining $(\ell-1)$ tasks. While in a nonpipeline processor with the same number of tasks and equivalent functions it takes

$$
\mathrm{T}_{1}=\ell \cdot \mathrm{n} \text { time delay. }
$$

The speedup $S_{n}$ of a pipeline processor over a nonpipeline processor is defined as,

$$
S_{n}=\frac{T_{1}}{T_{n}}=\frac{\ell . n}{n+(\ell-1)}
$$

The maximum speedup gain is $S_{n} \rightarrow n$, for $\ell \gg n$. In other words, with $n$ stages the maximum speed up that can be gained from a pipeline processor is n. However, this perfect speed up may not be achieved in practice due to memory conflicts, data dependency, program branch and interrupt operations.

### 1.3.2 Array Processors

Array processors can be defined as an array of interconnected identical processing elements (PE's). The PE's are controlled by a single control unit. Each PE consists of an arithmetic and logical unit ( $A L U$ ) and a local memory. Two essential reasons for building
array processors are firstly, economic for it is cheaper to build $N$ processors with only a single control unit rather than $N$ similar computers. The second reason concerns interprocessor communication, the communication bandwidth can be more fully utilised. A typical array processor is shown in Figure 1.10.


FIGURE 1.10: Array processors computer

The PE's are synchronized to perform the same function at the same time. The control unit decodes the instruction and broadcasts the instruction via control lines to all PE's simultaneously. The control unit can access information in both control or local memory. Each PE has access to its local memory only. Thus, a common instruction is executed by all PE's simultaneously using data from its local memory.

Different interconnection patterns between processors in array processors are used to permit data transfers between processors. In order to maximize the parallelism in an array processor, we must utilize as much of the available memory and processor bandwidths as possible. The array processor is eminently suitable for Linear Algebra. For example, if an array process contains $N\left(N=2^{n}\right)$ processor elements, the array $N \times N$ is stored by columns in such a way that each element of the matrix column is stored in the memory of the corresponding PE and one memory fetch transfers one column of the matrix into the vector of arithmetic units (PE). An example of an array processor is the ILLIAC IV computer. The general structure of the ILLIAC IV is shown in Figure 1.11.

The operational speed of an array processor is supposed to increase linearly as the number of processor elements (PE) is increased. However, this is not true due to interprocessor communication and data access overheads. The array processors can only be effective (i.e. maximum parallelism) if the array is completely filled with operands.

An associative store is used to overcome the bottleneck in enhancing the speed of conventional computers. An array processor using an associative type store as its memory is called an associative array processor. Array processors will be discussed again in a later section when SIMD type machines are described.


FIGURE 1.11: Structure of ILLIAC IV

### 1.3.3 Multiprocessor Systems

To improve system speed, reliability, throughput and availability multiprocessor systems were developed. A multiprocessor system is defined as a single computer with multiple processing units. It can also be defined as a system with more than one processing unit under integrated control. All processors share access to a common memory, input/output channels, control units and devices. Also, each processor
has its own local memory and private devices. For the complete multiprocessors system there is one integrated operating system which controls the hardware and software of the computer system. Processors are able to communicate between each other through the shared memories or through an interrupt network. A basic multiprocessor system is shown schematically in Figure 1.12.


FIGURE 1.12: Multiprocessor system

The interconnection subsystem is the main factor that characterizes the multiprocessor hardware system organization. Interconnection networks are surveyed in many references and among those are Jensen, J.E. and Baer, Jean-Loup 1976; Miller, J.S., Lickly, D.J., Kosmala, A.L., and Saponavo, J.A. 1970; and Noguchi, K., I. Ohnishi, and Morita, H. 1975 .

Enslow 1977, Hwang and Briggs 1984 identified three fundamentally different system orgenizations used in multiprocessors:

1. Time-shared comnon bus

The time-ghaxed common bus represents the simplest interconnection system for eithei sirgle or multiple processors. It consists of a common cowmaication path connecting all the functional units. These functionel units are a number of processors, memories, and input/output (I/O) devices. More than a single bus may be provided for throughput or relinbility reasons. The common bus architecture is characterized by its simplicity and low processor interconnection costs in adding or removing functional units, but a single failure in the bus halts the entixe system qualifying it suitable for small systems only. The system capacity is limited by the bus bandwidth and system performance may be degraded by adding new functional units.

The D.E.C., P.D.P. $-11 / 45$ and P.D.P. $-11 / 55$ computers are examples of mini-computers that are employed in this type of connection as shown in Figure 1.13. Different time-shared common bus interconnection schemes are shown in Figure 1.14.

a. The structure of PDP-11 CPU

b. PDP-11/45, PDP-11/55 system block diagram

a. Time-shared bus (single bus)

b. Time-shared bus (two one-way paths)

c. Time-share bus (multiple two-way bus)

FIGURE 1.14

## 2. Crossbar Switch Networks

To overcome the inadequacies of the time-shared bus organization, the crossbar switch is used. Interconnections between processors and memory units are increased in such a way that each processor is allowed to access individual memory unit, i.e., a separate path is provided for each memory as shown in Figure 1.15.

The main characteristics of the crossbar switch are high throughput, easy to isolate the malfunctioning device, the addition of functional units to attain improved system performance, and a most complex interconnection system. It is difficult to build large systems based on the crossbar switch concept due to the fact that the complexity grows at the rate of $O\left(n^{2}\right)$ for $n$ devices. An example of a crossbar interconnection system is the Carnegie-Mellon multi-mini processor (C.mmp) as shown in Figure 1.16.


FIGURE 1.15: Crossbar switch multiprocessor

3. Multiport memories

In multiport memory multiprocessors, the functions of control, priority arbitration, and switching between processors are centralized at the memory interface. To fulfil this, input ports for all processors are provided to each memory interface unit and response requests are controlled at the interface. It is possible to designate a portion of the memory as private to certain processors, I/O units, or a combination of both.

The main characteristics of such a system are expensive memory control, expansion from uniprocessor to multiprocessor system using the same hardware, system limitation by memory port design, and a large number of cables and connectors are required. Figure 1.17 shows a multiport memory system. An example of a multiport memory system is the IBM system $360 / 67$ as shown in Figure 1.18 .

a. The basic organization of a multiport memory system

b. Multiport memory organization including private memory


FIGURE 1.18: IBM system 360 Model 67


#### Abstract

1.4 ARCHITECTURAL CLASSIFICATION SCHEMES

Different approaches to the classification of computer architecture are suggested. Among those are, Flynn (1966) who proposed a classification scheme that is based upon instruction and data streams used in the system, while Shore (1973) based his classification on how the computer is organised from its constituent parts.


### 1.4.1 Flynn's Classification

Flynn classified computers into four classes according to the multiplicity of instruction and data streams. A stream is defined as a sequence of items (instructions or data) as executed or operated on by a processor. An instruction stream is a sequence of instructions as executed by the machine; a data stream is a sequence of data called for by the instruction stream. Flynn's four machine organizatiors as shown in Figure 1.19 are:

1. Single Instruction Stream Single Data Stream (SISD) computer which is the conventional serial computer (Von Neumann).
2. Single Instruction Stream Multiple Data Stream (SIMD) computer, (also known as array processors). These are made up of an array of processors, each executing the same string of instructions on different data.
3. Multiple Instruction Stream Single Data Stream (MISD) computer. This organization might be considered as unrealistic. There are more than one processing units, each receiving distinct instructions operating on the same data stream. Thurber (1976), Miklosko and Kotov (1984) considered that the pipelined systems fall into this category of computers.
4. Multiple Instruction Stream Multiple Data Stream (MIMD) computers are basically a network of $n$ processors connected together to provide a means for cooperating during a computation.

### 1.4.2 Shore's Classification

Shore presented a classification technique that derives machine descriptions from the description of a uniprocessor. Six different classes were identified which are shown in Figure 1.20 and are classified as:

1. Machine $I$ is a uniprocessor computer. Examples are the CDC-76 and CRAY-1 computers.
2. Machine II is the same as machine $I$ but with the addition of bitslice processing and access capability. Examples are the ICL DAP and STARAN computers.
3. Machine III is derived from machine II by adding parallel word processing and access capability. An example is the OMEN-60 computer.
4. Machine IV is derived from machine $I$ by replicating the processing units. An example is the PEPE computer.
5. Machine $V$ is derived from machine IV by adding interconnections between processors. An example is the ILLIAC IV computer.
6. Machine VI is derived from machine $I$ by distributing the processing logic throughout the memory. It is called a logic-in-memory array (LIMA) processor. Examples are the associative memories and associative processors.

From the above classification two main parallel computer classes emerge,
the SIMD and MIMD computers and these will be discussed in more detail.

The parallel systems installed at Loughborough University will also be discussed. Finally, Data Flow computers and VLSI models of computation will be discussed in later sections.

IS

a. SISD computer

CU : control unit
PU: processor unit
MM: memory module
SM: shared memory
IS: instruction stream
DS: data stream

b. SIMD computer

c. MISD computer

FIGURE 1.19: Flynn's classification

d. MIMD computer

FIGURE 1.19(cont.) Flynn's classification


1. Machine I

2. Machine II

3. Machine III

4. Machine IV

FIGURE 1.20: Six machine classes defined by Shore (1973).

5. Machine V

6. Machine VI

### 1.5 SIMD MACHINE

The SIMD type machine or the array processor (see Section 1.3.2) consists of N processing elements (PEs) under one control unit (CU). Parallelism of the system is achieved by multiple processing units. Through the availability of scalar and vector operation SIMD computer programming is considered to be simple. A whole vector of data can be executed in one instruction operation. In a SIMD type machine the same operation is performed at the same time over data in all processing elements. Two SIMD configuration types are shown in Figure 1.21. Type a shows N synchronized PE's where all the PEs are under one control CU. Each $\mathrm{PE}_{i}$ has a local memory $\mathrm{PEM}_{i}$ and the control unit has its own main memory. User programs are loaded into the CU. The CU decodes the instructions and decides where the instructions should be executed. Vector instructions are broadcast to the PEs while scalar or controltype instructions are executed inside the Cu . An example of this type of configuration is the ILLIAC-IV computer.

Type b shows another configuration which consists of N PEs and P memory modules. This configuration differs from configuration a in two ways. Firstly, local memories are replaced by parallel memory which are shared by all the PEs. Secondly, the inter-PE permutation network is replaced by the inter-PE memory-alignment network, which is controlled by the CU. An example of this type of configuration is the Burroughs Scientific Processor (BSP).

One of the major issues in the design of SIMD computers is the interconnection and transfer of data between the PE's. Different interconnection networks have been proposed for SIMD computers. Obviously,

a. Configuration a (Illiac IV)

b. Configuration b (BSP)

FIGURE 1.21: Architectural configuration of 2 SIMD array processors
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. Hwang and Briggs [1984] classified a SIMD interconnection network into static networks and dynamic networks. Static networks are classified according to the dimensions required for layout. For example, one-dimensional, twodimensional, three-dimensional, and hypercube as shown in Figure 1.22. Dynamic networks are classified into single-stage networks and multistage networks as shown in Figure 1.23.

- To run a program efficiently, it is necessary to match the algorithm requirements with the interconnection pattern concerned in order to prevent extra communication delays that may increase the execution time and reduce the speed-up factor.

Associative memory has been used to overcome the limitation gain in speed of conventional computers due to the physical separation between data storage and processing units. In associative memory data can be retrieved using their content or part of their content. The major advantage of associative memory over conventional random-access is its capacity of performing parallel search and parallel comparisons. Another class of SIMD computer have been built using associative memory instead of conventional random-access memory.

An associative processor is an SIMD machine with the following properties: (1) stored data items are content-addressable and (2) arithmetic and logic operations are performed over many sets of arguments in a single instruction. Figure 1.24 shows a block diagram of the associative computer. The basic memory element of the


(e) Near-neighbour mesh

(h) Chordal ring

(j) 3-cuke-connected cycle

FIGURE 1.22: Static network: (a) one dimensional; (b-f) two dimensional; and ( $g-j$ ) three dimensional

(b) $8 \times 8$ baseline network
(a) $8 \times 8$ shuffle-exchange (multistage) (single stage)

(c) $8 \times 8$ Benes network (multistage)

FIGURE 1.23: Dynamic network


FIGURE 1.24: Block diagram of the associative computer
associative memory is called the bit-cell. In order to retrieve stored data items by their content or part of their content, all cells receive simultaneously the required word $C$ and the mask M. A memory cell is regarded selected if the condition $(C=W)_{\wedge} M$ is satisfied for all bits.

The comparison process of the associative memory is the dominating factor that classifies the architecture of associative processors. Associative processors are classified into four categories as:

- fully parallel,
- bit-serial,
- word-serial,
- block-oriented.

There are two types of fully parallel associative processors: wordorganized and distributed logic type. In the word-organized type, the comparison logic is associated with each bit cell of every word and the logical decision is available at the output of every word. In the distributed logic type the comparison logic is associated with each character-cell or with a group of character-cells. In a bit-serial associative processor, only one bit column (also called bit-slice) of the whole word is operated upon at a time. A word-serial associative processor is essentially a hardware implementation of a simple program loop for search. A block-riented associative processor can be implemented by using a logic-per-track rotating memory which consists of a head-per-track disk with some logic associated with each track. The two most important categories are the fully parallel and the bitserial associative processors. The Parallel Element Processing Ensemble (PEPE) and STARAN are the best-known fully parallel and bitserial associative processors respectively.

In a comparison of the associative SIMD processor with a SIMD array processor, the associative processor has the following characteristics, i.e.,

- it allows memory addressing down to the bit level,
- the word length can be chosen arbitrarily,
- due to the low cost of integrated circuit, the number of PEs can be large in both,
- arithmetic units of single memory cells work serially bit-by-bit, but they perform simultaneously the same operation which is assigned by the central control unit to all cells.


### 1.6 MIMD COMPUTERS

Multiple instruction stream multiple data stream (MIMD), computers include machine organizations usually referred to as "multiprocessors" (see Section 1.3.3). The MIMD computer can be considered as a collection or network of minicomputers or microcomputers and collectively as a multiprocessor system. The MIMD computer consists of multiple processors, each processor generating its own instruction stream which it executes on its own data stream. These processors are connected either through a shared memory or via high-speed or low-speed data links.

Figure 1.25 shows an MIMD structure consisting of $P$ memory modules, n processing units, and m input/output channels. Different interconnection networks are shown in Figure 1.25, the processor to I/O interconnection network enables the connection of the I/O channels to any processor. The processor to memory interconnection network enables the connection of a processor to any memory unit. The processor to processor interconnection network is in fact an interrupt network rather than a data exchange network, since the data exchanges can be done through the memory to processor interconnection.

Memory conflict and processor interconnection are the main two factors that degrade system performance. To reduce these two problems, a "private memory" is recommended to be associated with each processor in which its important data is stored.

MIMD systems may be classified into tightly coupled systems or loosely coupled systems. Tightly coupled systems are characterized by:


Input/Output
Channels

1. Shared main memory used as an interconnection means between processors, and hence all the processors can access all the memories and execute code out of them.
2. Input/output and other system resources are shared by the processors.
3. The interprocessor communication is of the order of the bandwidth of the memory.
4. Synchronization between cooperating processors will be required.
5. A small local memory or high-speed buffer (cache) may exist in each processor.
6. The connection between the processors and memory is done either by a multiported memory or by inserting an interconnection network between the processors and the memory.

The major limitations of the performance of tightly coupled multiprocessor systems are:-

1. The degradation in performance due to conflicts to access the main memory or the input/output devices.
2. The delays due to synchronization and scheduling of jobs on the different processors.
3. The choice of the processor-memory interconnection network.

Different approaches in the tight coupling of MIMD systems are shown in Figure 1.26.

The main characteristics of loosely coupled systems are:-

1. Each processor in the system has its own memory, that is they do not share a common memory.
2. An explicit communications interface between the processors are needed.
3. Concurrent processes may be performed asynchronously.

a) Shared bus approach

b) Multiported shared memory approach

c) Crossbar interconnection approach
4. Each processor can stand by itself with its own storage.

In loosely coupled systems, normally, one of the processors is designated as overall system control (global processor). The other processors are called local processors. All jobs enter the system through the global processor. If a global processor fails, one of the locel processors may act as global processor.

In loosely coupled systems, synchronization, task partitioning, software control, and commination data transfers are the problems to be taken into consideration. To improve performance, the user must determine how to divide the task between the computers so that they can operate in parallel.

Two variations of loosely coupled systems are shown in Figure 1.27.
A tightly coupled multiprocessor has a distinct performance advantage over the loosely coupled multiprocessor, and is a good general solution because all its resources are shared and directly accessible, and can be accessed and allocated faster. A shared memory also offers the quickest way to pass data between CPUs.

a) Using high-speed bus

FIGURE 1.27: Loose coupling in MIMD systems.

b) Using a slower link

FIGURE 1.27: Loose coupling in MIMD systems (continued)

Flynn, et al (1970) suggested an alternative approach, for the design of the MIMD computer. He proposed to interconnect several independent processors, each of which executes an independent instruction stream. The proposal is to convert the processors into "skeleton" processors, by removing from them all the arithmetic functions and computational logic. These functions are performed by highly specialized high-speed processors as shown in Figure 1.28. The resulting system avoids many of the connection problems associated with shared resource systems.

Consider for example, the events that occur when a skeleton processor generates an ADD instruction. After obtaining the operands
for the instruction, the processor requests access to a high-speed adder. If one is available, the operation is performed and the result returned to the skeleton processor. In the case of conflict, the request for computation can be queued or the request can be repeated until an adder is available.


FIGURE 1.28: MIMD computer with skeleton processors and centralized computation facilities

In a multiprocessor system, an interconnection device between the processors and memory modules, and between the processors and I/O subsystems are needed to give the processors in the multiprocessor system the ability to share both the main memory modules and $1 / O$ devices.

Enslow [1970] and [1974] classified multiprocessor interconnections into three basic types, these types are:

1. Time shared or common buses
2. Crossbar switch matrix
3. Multiported memories.

The main advantages of the MIMD computers are high throughput and greater reliability. High throughput can be achieved by dividing the processes into many subprocesses which can run on different processors concurrently. While greater reliability is achieved by easily isolating the faulty resource, (processors and memory modules) which are generalpurpose resources, and thereby achieving a better fault tolerance level.

MIMD computers are more general-purpose in application than SIMD computers. The processors in MIMD computers need not be synchronized instruction-by-instruction as in the SIMD computers. However, it is required that the processing algorithms exhibit a high degree of parallelism, so that several processors are active concurrently at any time.

In MIMD systems, it is not generally true in practice that $n$ processors should give $n$ times the throughput of a uniprocessor. This is due to the overheads needed to coordinate the activities between the cooperating processors. The main difficulties that arise in MIMD computers are the partitioning strategy, i.e., identifying parallelism in processing algorithms to invoke concurrent processing streams. Also, the interconnection network design, interconnection between the processor-to-memory or processor-to-I/O devices is the most expensive component of the system and can become a bottleneck.

Examples of some multiprocessor systems are, C.mmp, C.m* (both research machines constructed and developed at Carnegie Mellon University), S-1 system (currently under development at Lawrence Livermore National Laboratory), and many commercially available multiprocessor systems, including some models in the IBM 370 series and 3080 series, the Univac 1100 series, the Tandem Nonstop system, the HEP, and the Cray X-MP.

Finally, in more detail some of the above examples will be described.

The first system, the C.mmp system, as shown in Figure 1.29 is an MIMD system developed by Carnegie-Mellon University during the years 1971-1978. There are 16 memory blocks (MO-M15) connected to 16 processors (PO-P15) through a 16-by-16 crossbar network. Each processor has a local memory block (Mlocal), a disk unit, and other peripherals. An interprocessor bus which connects the entire set of processors is used to perform the general function of interprocess communication. The bus provides common clock information as well as interprocessor interrupts. Each processor is a model of the D.E.C. PDP Il. Some modifications were required to make these processors suitable for a multiprocessor environment and to provide software protection. The primary memory consists of 1.4 Mbytes of core memory (eleven partitions) and 1.3 Mbytes of MOS memory (five parts). The core memory in each partition consists of eight modules of 16 K bytes each; thereby providing eight-way interleaving. The MOS memory is configured as four 65 K bytes modules per partition. These modules are not interleaved. The local (non-shared) memory of each processor is an 8 K byte core memory. A


FIGURE 1.29: Structure of the C.mmp multiprocessor
standard unibus interface is used to connect peripheral devices to each processor. The DMAP unit maps the address on the unibus into the address required for primary memory access. All references to the shared memory are first checked against the contents of the 2 K bytes cache. If data are available in the cache, the primary memory is bypassed. Thus, cache memory is employed to reduce the memory contention rather than to speed up the memory access. The crossbar switch allows the maximum concurrency of sixteen paths when all processors request different memory parts. It also resolves the memory contention when more than one processor requires the same memory location. The C.mmp system is controlled by the Hydra operating system. This system has been used as a testbed for parallel algorithm design and has contributed to reliability and software recovery problems.

The second system is the $S-1$ multiprocessor system which can be described as a high-speed general-purpose multiprocessor. The $\mathrm{S}-1$ is implemented with the $\mathrm{S}-1$ uniprocessor called Mark IIAs. Figure 1.30 shows the logical structure of the $\mathrm{S}-1$ multiprocessor. This structure consists of 16 independent Mark IIa uniprocessors which share 16 memory banks via a crossbar switch. Each processor has a private cache which is transparent to the user. A diagnostic processor is connected to each uniprocessor, crossbar switch and memory bank. This diagnostic processor can probe, report and change the internal state of all modules that it monitors. Each memory bank can contain up to $2^{30}$ bytes of semiconductor memory and hence a total physical address space of 16 gigabytes $\left(2^{34}\right)$. The large memory addressibility of the $S-1$ essentially eliminates the programming cost associated with managing multiple types
of computer system storage. The crossbar switch is designed to provide access to multiple memory requests. The crossbar also handles interprocessor communication. The S-l multiprocessor system has the capacity of using dual crossbar switched for reliability and a front end (diagnostic-maintenance) processor to remove a failing switch and substitute an alternative switch. The memory controllers control all the read/write accesses between different uniprocessors by using send and receive messages via interprocessor-interrupt mechanisms within the crossbar switch. The S-l design provides the I/O subsystem which consists of many microcoded I/O channels. Each channel is managed by an I/O processor. The I/O subsystem also contains I/O buffers or memories which are accessible as part of the $s-1$ processors' address space. Each I/O peripheral processor may be connected to input-output ports on at least two uniprocessors, so that the failure of a single uniprocessor does not isolate any input-output device from the multiprocessor system. The performance of each Mark IIA is achieved by extensive pipelining due to advances in microcode: programming, hardware structure, and implementation technology.

The Mark IIA processor consists of five major components as shown
in Figure l.3l. These components are extremely fast, relatively special-purpose programmable controllers that operate in parallel to provide a high performance. Four components that form the instruction pipeline are for instruction fetch (F sequencer); instruction decode ( $P$ sequencer); operand operation (I sequencer) and arithmetic execution (A module). These sections are internally pipelined to achieve a maximum instruction-issue rate of one instruction per 50ns, which is equivalent to a maximum data throughput rate of 720 million bytes.



FIGURE 1.31: The internal logical structure of the S-1 Mark IIA uniprocessor

For the $\mathrm{S}-1$ multiprocessor system, there exists a single user operating system, multiu-user operating system, and advanced operating system. The single-user operating system is a simple stand-alone system which runs a single task at a time and provides only basic I/O functions. The maltiuser operating system to be developed will be based on the Unix operating system because it is a small, relatively powerful system and has demonstrated a suitability for transport. The advanced operating
system for the $S-1$ is the full functionality system Amber. The Amber operating system supports a mixture of applications which include a real-time system (e.g. signal processing), interactive use (e.g., physical simulation) and secure environment for data. The Amber operating system combines the functions of the file system and virtual memory. It also supports multitasking by the division of problems into cooperating tasks.

The final example of a multiprocessor system is the HEP system. The Heterogeneous Element Processor (HEP) is a large-scale scientific multiprocessor system and is the first commercially available MIMD multiprocessor system. The system contains up to 16 process execution modules (PEM) and up to 128 data memory modules (DMM). The PEM's or DMM's are connected with the I/O and control subsystem via a high-speed switching network. The PEM is the computational element of the HEP. Figure 1.32 shows an example configuration of the HEP with 28 switching nodes, four PEMs, four DMMS, a mass-storage subsystem, an I/O control processor, and a node connection to four other devices. The massstorage subsystem consists of three major components. A large MOS buffer memory provides an I/O cache function to mask the seek and rotational delays of the disks. Disk storage modules provide storage increments of 600 megabytes. I/O channels couple the disk storage modules to the I/O cache and are controlled by the I/O control processor. Figure 1.33 illustrates the components of the mass-storage subsystem. The system can handle up to 32 I/O channels, with each channel supporting a transfer rate of up to 2.5 megabytes/s. Each disk storage module consists of two disk drives. The HEP switch is a synchronous, pipelined, packet-switched network consisting of an
arbitrary number of nodes. Each node, which consists of three full duplex ports, is connected to its neighbours. These neighbours may be PEMs, DMMs, subsystems, or other nodes. Each node switch is programmed to determine the best output port routing to the final destination. Such programmed routing techniques allow for alternative routing to bypass a faulty component.


FIGURE 1.32: The architecture of a typical HEP system with four processors

## Switch network



TO I/O and control subsyst subsystem

## I/O choice <br> Memory



FIGURE 1.33: The mass storage system (MSS) in the HEP

The execution of multiple independent instruction streams on multiple data streams is implemented by replicating the functional units in each PEM. Maxmial parallelism in the HEP system is achieved by providing multiple independent instruction streams executing multiple data streams in a pipelined execution environment as shown in Figure 1.34.

Each PEM consists of its own program memory and an instruction processing unit (IPU). Up to 50 instructions may be in various stages of execution operating on one or more data streams simultaneously.

There are many applications which can be run on the HEP machine. A variety of applications are: traditional multiprogramming, solution of large-scale systems of ordinary and partial differential equations, but the greatest potential is in the simulation of a discrete event system or process driven simulation.


FIGURE 1.34: Achieving maximal parallelism with replicated hardware in the HEP (MIMD processing)

### 1.7 LOUGHBOROUGH UNIVERSITY PARALLEL SYSTEMS

Two MIMD type parallel systems have been developed at the Department of Computer Studies of Loughborough University, these systems are the Interdata Dual System and the more powerful NEPTUNE system.

The first system consisted of two identical Interdata model 70 processors. Each processor has 32 Kb of private memory and shares a 32 Kb shared memory as shown in Figure 1.35


FIGURE 1.35: Interdata Dual System

In this system description the two processors will be referred to as A and B respectively. The system had the asymmetrical property that 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, also 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. On the other hand, processor A has a minimum delay of one microsecond in memory accession, while processor $B$ has zero microsecond as the minimum delay. The performance measurements of this Interdata Dual system has been presented by Barlow and Evans (1977).

The second MIMD type computer which has been developed at the Department of Computer Studies of Loughborough University is the NEPTUNE system (Barlow, et al 1981). This system is based upon the Texas Instruments 990/10 minicomputer. The current configuration is shown in Figure 1.36. The system contains five linked busses (TILINES) and four processors (numbered as PO,P1,P2, and P3 respectively). Each one of the four processors are attached with the TILINE acting as the local bus for that processor. Each processor can also access its own (private) memory via its local TILINE. The current size of private memory for processors P1, P2 and P3 are 192K bytes and for processor PO it is 384 K bytes (the maximum size of private memory for processors PO,Pl,P2 and P3 are $576 \mathrm{~K}, 384 \mathrm{~K}, 576 \mathrm{~K}$ and 320 K bytes respectively). Processor PO has a 10 Mbyte disc drive on its local TILINE, while processor P2 has a controller with a 474 Mbytes Winchester disc drive and a tape streamer attached. Each of the local TILINE's is attached via a TILINE coupler to the fifth (shared) TILINE. To this shared TIIINE there is attached 104 Kbytes (rising to 184 Kbytes) of memory and a 50 Mbytes disc.

Each processor can access a minimum of 232 Kbytes. This is because, the TILINE coupler is arranged so that the shared memory follows continuously from the local memory of each processor. The 50 Mbytes disc can be accessed by all the processors which will receive the disc interrupts.

In addition, the NEPTUNE system can operate as four individual processing systems.

The Texas Instruments $990 / 10$ minicomputer runs under the DXIO operating system which is a sophisticated multi-tasking system and supports a tree-structured filing system. Modifications have been made to the DXIO to allow parallel processing to take place and to permit various management policies for the shared resources to be investigated.

The shared memory storage can be claimed by any processor, and once it is granted this storage behaves as if it was local memory. A small area on top of the shared memory is reserved for managing interprocessor cooperation. A parallel program to be run on the four processor system logically consists of two parts. The first part contains the program code and local variables, while the shared variables are stored in the second part. When a processor receives a request to execute a parallel program (or, more correctly, a parallel task), shared memory space is claimed by that processor and the segment containing the shared variables is loaded into this space. The management area in common memory is set to contain pointers to the shared segments and tasks are activated in other processors with sufficient information to enable them to run the requested program. The non-shared segment(s) are loaded into the private memory of each
processor. The tasks running on all four processors have the right to access simultaneously the shared disc and the files stored on it by the DX10.

To realise parallel operation, more than one processor can create, open, access (read and write) and delete files. There are however, two restrictions on simultaneous access to a single file from more than one job/task; the first is that it is not possible to have two tasks to 'open' a file for writing, the second is, if two tasks on different machines have a file open one for reading and the other for writing then if the writer changes the size of the file the reader will not be informed until the writer closes the file.

When a user logs on, a task is initiated on his behalf. This task provides the user interface to the system and is called the System Command Interpreter (SCI). Commands may be issued in different ways by the SCI. A series of menus are the simplest level that will display a sequence of command classes and eventually arriving at a list of commands. All the commands are implemented either as tasks running under the operating system (e.g. compilers and utility programs) or as functions of the SCI (those corresponding to a supervisor call). In the NEPTUNE system, the foreground and background modes are available for a task to be run. Using the terminal, only one foreground task may be executed at any one time. An interactive program must be run in the foreground. Several tasks and commands may be executed in the background. Background tasks should not involve $I / O$ with the terminal. While a background task is executing, the SCI is still running and available to process user requests. Commands are available for inspecting the state of the background tasks.

The access time to both shared memory and local memory for the four processors in the NEPTUNE system are not the same. The times for local memory access are: $0.98 \mu \mathrm{sec}, 0.95 \mu \mathrm{sec}, 0.92 \mu \mathrm{sec}$ and $0.92 \mu \mathrm{~s}$ for processors PO, P1, P2, P3 respectively. While the time for shared memory access are $1.73 \mu \mathrm{sec}, 1.70 \mu \mathrm{sec}, 1.68 \mu \mathrm{sec}$ and $1.68 \mu \mathrm{sec}$ for processors PO, P1, P2, P3 respectively. Although the processors are identical in many hardware features, they are also different in their speeds.

The hardware features and the operating system of the NEPTUNE system have now been discussed. The programming concepts of the NEPTUNE system will now be discussed in the next chapter.


FIGURE 1.36: The current NEPTUNE configuration

### 1.8 DATA-FLOW COMPUTERS

The computer architectures discussed in the previous sections are known as control flow (Von Neumann) machines. In conventional Von Neumann's (known as Control Flow, CF) computer, the program is stored in the memory as a serial sequence of instructions. Computations in CF computers are done according to the flow of control in the program. It is not possible to execute any instruction until all the previous instructions in the program have been executed, i.e., if there exists an instruction in a program such that the data is available and could be executed immediately, it is not executed until its turn comes in the program. This is one of the main difficulties in the utilization of the natural parallelism of algorithms in the CF model of computation. Another architectural model for computer systems is created to make it possible to express the natural parallelisms of algorithms, this model is the data flow model of computation, also known as a data-driven system. In a Data Flow (DF) computer, the course of computation is controlled by the flow of data in the program. That is, an operation is performed as and when its operands are available. The sequence of operations in the DF computer obey the precedence constraint imposed by the algorithm used rather than by the location of the instructions in the memory. In DF machine it is possible to carry out in parallel as many instructions as the given computer can execute simultaneously. After executing the instruction, the result is distributed to all subsequent instructions which make use of this partial result as an operand. In this way, the DF model of computation exploits in a simple manner the natural parallelism of algorithms. In computer architecture, this makes it possible to create systems which can dynamically adapt
their inner configuration to the natural structure of the algorithm being performed.

As an illustration of DF computation, the computation of the roots of a quadratic equation is shown in Figure 1.37. Assuming that $a, b$ and $c$ values are available, (-b), ( $b^{2}$ ), (ac), and (2a) can be computed immediately, followed by the computation of (4ac), ( $\left.b^{2}-4 a c\right)$ and $\sqrt{\left(b^{2}-4 a c\right)}$, in that order. After this, $\left(-b+\sqrt{\left.b^{2}-4 a c\right)}\right.$ and $\left(-b-\sqrt{\left.b^{2}-4 a c\right)}\right.$ can be simultaneously computed followed by the simultaneous computation of the two roots. The only requirement is that the operands be available before an operation can be invoked.

The two basic models of data flow computer architecture which are designed by Miller and Cocke (1972) are the:

1. Search mode configurable computer (SM type)
2. Interconnection mode configurable computer (IM type)

Both models are characterized by the possibility of dynamic adaptation of its configuration to the structure of algorithms. This is done by interconnecting (according to the graph) the processors that correspond the fye operators in the data flow program of the problem. Reconfiguration is done either by hardware or software means. In the IM type, the interconnection of processors is actually implemented through a large switch, i.e., by hardware means. In the SM type, the interconnection of processors is simulated by using a special instruction format, i.e., by software means. Due to reconfigurability, the data flow computer is able to achieve the same performance as a specialized system, whilst still keeping its general purpose capabilities.

Figure 1.38 shows the search mode (SM) type computer which consists of a memory, a functional unit, and a control unit (searcher).


Root 2

FIGURE 1.37: A data-flow graph for the computation of the roots of a quadratic equation


FIGURE 1.38: Search mode configurable computer

The functional unit is composed of a certain number of processors. A searcher is a specialized unit for generating tasks for processors which belong to the functional unit. The memory is used to store data and instructions, either together or separately. A free processor asks the searcher for a task. The searcher locates a suitable task in the memory or composes it from various components stored in different parts of the memory and sends it to the selected processor in the functional unit for execution. Processors may be adders, multipliers, conditional tester, I/O processors, etc. Because the searcher performs more than half of the work necessary in traditional computers for the execution of an instruction, the performance of a computer will depend mainly on the throughput of the searcher which the effective utilization of the processors largely depends upon. Also the performance of this computer will depend on the type and number of processors, and memory speed. Figure 1.39 shows the interconnection mode configurable computer (IM) type. A computer of $I M$ type can be reconfigured to adapt as much as


FIGURE 1.39: Interconnection mode configurable (IM) computer
possible to the algorithm being executed. This is done through an interconnection network (switch) as shown in Figure 1.39. Since it is not possible to execute the whole program at one time on a set of processors, the data flow program is divided up into a number of blocks taking into consideration the number of processors available at any one time. This is done by a compiler which determines the interconnection of processors, so that the computation structure created corresponds to the graph of the data flow program or some part of it. This interconnection is encoded and stored in memory as a set-up instruction for the switch. This set-up instruction is fetched first and sent to a set-up control which does the interconnection of the procesșors. In this way, the computation structure is ready to execute the computation of a block of the program. After processing a block, the processors involved and a part of the switch are released and can be used for setting up the structure of another block.


#### Abstract

Several data flow machines have been built using different architectures. Currently, data flow computers are operational at different places in the United States, Japan and Europe. Among those operational data flow computers there are those at the University of Utah, Manchester, Toulouse, and two different projects at M.I.T.

One of the first data flow computers was introduced by Dennis and Misunas [1974, 1975] as shown in Figure 1.40. The structure of this data flow computer consists of five major sections connected by channels through which information is sent in the form of discrete tokens '(packets). The memory section consists of instruction cells which hold instructions and their operands. The processing section consists of processing units that perform functional operations on data tokens. The arbitration network delivers operation packets from the memory section to the processing section. The control network delivers a control token from the processing section to the memory section. The distribution network delivers data tokens from the processing section to the memory section.

Instructions held in the memory section are enabled for execution by the arrival of their operands in data tokens from the distribution network and control tokens from the control network. Enabled instructions, together with their operands, are sent as operation packets to the processing section through the arbitration network. The results of instruction execution are sent through the distribution network and the control network to the memory section, where they become operands of other instructions. Each instruction cell consists of three registers. The first register contains the instruction code and all conditional control data, with operands in the second and third registers.


The arbitration network provides a path from each instruction cell to each processing unit and sorts the operation packets among its output ports according to the operation codes of the instructions they contain. For each operation packet received, a processing unit performs the operation specified by the instruction using the operand values in the packet and produces one or more result tokens, which are sent to the instruction cells through the control network and the distribution network. Each result token consists of a result value and a destination address derived from the instruction being processed by the processing unit.

The functions performed by the processing unit are distributed among several sections of the data flow processor. The transmission of packets over each channel used an asynchronous protocol so that the five sections of the computer can operate independently without using central timing signals. The instruction cells are assumed to be physically independent, so that at any time many of them may be enabled. The arbitration network should be designed to allow many instruction packets to flow through it concurrently. Similarly, the control network and the distribution network should be designed to distribute dense streams of control and data packets back to the instruction cells. In this way, both the appetites of pipelining and parallelism are satisfied.


FIGURE 1.40: The data flow computer architecture

### 1.9 VLSI SYSTEMS

Due to the current development in hardware technology, Large Scale Integrated (LSI) electronic circuitry has become so dense that a single silicon LSI chip may contain tens of thousands of transistors. These hardware advances have led to more functions being implemented in hardware, e.g. it has been possible to implement a sophisticated 16bit processor on a chip. The actual number of components in one chip depends on the speed of the devices and the regularity of the patterns used to lay them out on the chip.

As LSI technology advances, Very Large Scale Integrated (VLSI) circuit designs were introduced in which the number of transistors that the LSI circuit will contain will be increased by another factor of 10 to 100 in the next decade (Mead and Conway (1980)). By late 1980's it will be possible to fabricate chips containing millions of transistors. As we move into the VLSI era, 32-bit processors with memory and input/ output support will also be available on a chip. In addition, it will be possible to implement SIMD and MIMD architectural designs on a chip.

The key factors of VLSI technology are: its capacity to implement enormous numbers of devices on a chip, low cost and high degree of integration while the main VLSI problem is to overcome the design complexity. One solution to reduce the complexity of a VLSI chip is to use a regular design structure (patterns) as in a memory chip.

In order to get full use of VLSI capabilities, computer-aided design methodologies, and design systems are needed for chip design. The study of VLSI circuits is not limited to 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 communications techniques to exploit the potentiality of the VLSI system and its applications into highly parallel and specialised computers. An excellent 'state of the art' survey can be found in the January 1982 issue of IEEE Computer magazine.

The separation between the processor from its memory and the limited opportunities for concurrent processing are the main difficulties in the conventional (Von Neumann) computers. VLSI offers more flexibility than conventional (Von Neumann) computers to overcome these difficulties because memory and processing architectures can be implemented with the same technology and close proximity. The potential power of VLSI has to come from the large amount of concurrency that it may support. The degree of concurrency in a VLSI computing structure is largely determined by the underlying algorithm. Enormous parallelism can be obtained by introducing a high degree of pipelining and multiprocessing while designing the algorithm. The requirements of parallel architectures for VLSI have been discussed by many authors among those are Kung (1982) and Seitz (1982). The design should contain a few modules which are replicated many times (i.e., simple and regular) and using both pipelining and multiprocessing principles. Finally, a successful algorithm for VLSI design will be the one where the communication is only between neighbouring processors.

One way of achieving parallelism is by attaching a special-purpose parallel processor to the system bus of a microcomputer to speed-up the more computationally bound tasks, as shown in Figure 1.41. The special-purpose processor, for example, could be designed to exploit


FIGURE 1.41: Microcomputer with attached special-purpose processor the features of a particular class of problems (e.g. the finite element machine (Podesiadlo and Jordan (1981)) or alternatively it could be designed for a general computational task, like the solution of linear equations, as envisaged by Kung (1979).

There are two general architectural designs of the attached parallel processor to be considered in more detail in this section. The first is the multiprocessor lattice architecture based on the idea of several processing elements operating under a centralised control and the second is a systolic array architecture which makes extensive use of pipelining.

Dew, Buckley and Berzins (1983) defined a multiprocessor lattice architecture as, an $N \times N$ array of processing elements which execute concurrently under a centralised control and transmit along local communication path connecting neighbouring processing elements. Each processing element has a private memory to store both results and also temporary values which may be needed. A global bus may be used by the processing elements to communicate between each other but this is not an essential feature of the architecture. The ICL DAP computer is an
example of an array lattice architecture.
Another example is the Configurable Highly Parallel Computer (CHIP) (Snyder (1982)) where programmable switches are provided between the processing elements. Because the switches are programmable this means that the lattice is reconfigurable dynamically. In this type of architecture, a significant amount of software support is needed to program each processing element by the host computer. The need for a large local memory, together with the need to broadcast a program to each element means that the architecture is less well suited than the systolic architectures to the requirements of a VLSI system. Still by the use of VLSI systems it will drastically cut the cost of building the hardware.

The second architecture, the systolic array architecture was developed by Kung (1982) in which the wiring in a chip design is reduced to a minimum. A systolic system consists of a set of interconnecting cells (processing cells), each capable of performing a "hardware" simple arithmetic operation.

The structure gives a simple and regular pattern that allows for easy communication between the cells. Information in a systolic system flows between the cells in a pipelined fashion, where communication with the outside world occurs only at the boundary cells, i.e. boundary cells may be I/O ports for the system. Figure 1.42 illustrates the basic principle of a systolic array. By replacing a single processing element with an array of P.E.'s, a higher computational throughput can be achieved without increasing the memory bandwidth. Data items are transmitted from the memory through the array of P.E.'s and can be processed effectively at each cell it passes. This is possible for a

a) The conventional processor

b) A systolic processor array

FIGURE 1.42: The concept of systolic processor array
wide class of computer-bound computations where multiple operations are performed on each data item in a repetitive manner.

As an example, if each PE in Figure 1.42 operates with a clock of 100 ns . The conventional memory-processor organization in Figure
1.42a has at most a performance of 5 million operations per second. With the same clock rate, the systolic array will result in 30 MOPs performance. This gain in processing speed can also be justified with the fact that the number of pipeline stages has been increased six times in Figure 1.42 b .

The advantages of the systolic approach are, the ability to use each input data item many times, modular expansionability, simple and regular data and control flows. Figure 1.43 shows different VLSI systolic array structures for different computer-bound algorithms. These computations form the basis of many signal and image processing, matrix arithmetic and database algorithms. The major problem with systolic array is still in its I/O limit. The globally structured systolic array can speed-up computations only if the I/O bandwidth is high.

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.

(a) One-dimensional linear array

(b) Two-dimensional square array

(d) Binary tree

(c) Two-dimensional hexagonal array

(e) Triangular array

FIGURE 1.43: Various systolic array configurations

## Chapter 2

PARALLEL PROGRAMMING PRINCIPLES

### 2.1 INTRODUCTION

As seen in Chapter 1 the recent advances in hardware technology and computer architecture leads to faster and powerful parallel computer systems. Problems for parallel computer systems require some extra. programming facilities which come under the heading of parallel programming, to distinguish it from the conventional programming of single-processor computers. The two new concepts behind the new ideas of parallel programming theory are parallelism and asynchronism of programs. Gill (1958) defined parallel programming as the control of two or more operations which are executed virtually simultaneously, and each of which entails following a series of instructions.

There is a gap between the hardware architectual advances and the development of programming languages or software production tools to utilize these technological and architectural advances. Any parallelism in an algorithm will be lost when it is expressed in a sequential highlevel language, so this type of language is not entirely suitable for parallel computers. A parallel language which provides the programmer with sufficient tools to enable the construction of efficient algorithms and at the same time effectively utilize the hardware is needed. It is now apparent that parallel processors required a language created in their own generation using in so far as is possible the experience accumulated in language design and implementation technique and which incorporates the new features that are necessary in writing algorithms for these machines.

Most of the high-level languages currently used to program
parallel computers are extensions of languages which were specifically designed many years ago, for sequential machine architectures. Examples
are, Fortran-like languages, CFT for the Cray-l (Russell, (1978)), and IVTRAN for the ILLIAC-IV (Millstein (1973)). An algol-like language, Glypnir (Lawrie, (1975)) for Illiac-IV was also developed. Many parallel programming languages have been proposed for different types of machines. For example, ACTUS (Perrott, (1979)) has been designed to enable the specification of parallelism directly and has been used to exploit parallelism in algorithms implemented on array processors such as the ILLIAC-IV. Other languages such as TRANQUIL (Abel, et al (1969)), an Algol-like language and the experimental VECTRAN (Paul and Wilson (1975)) language for vector/matrix array processing have also been proposed.

Perrott (1979) has classified the available or proposed parallel languages into three categories:-

1. Detection of problem parallelism, in which the programmer constructs a problem solution in a sequential programming language and a compiler tries to detect any inherent parallelism.
2. Expression of machine parallelism in which the syntax of programming language reflects the underlying parallelism of the hardware either directly or by means of subroutine calls.
3. Exploitation of problem parallelism in which the program and data structures enable the programmer to directly express the parallel nature of a problem.

### 2.2 HIERARCHIES OF PARALLELISM

Both Task and Process are intended to mean a self-contained portion of a computation that once initiated can be carried out to its completion without the need for additional inputs. The completion of a task is significant in that its occurrence can initiate the execution of other sets of tasks. The tasks $T_{1}, T_{2}$ and $T_{3}$ of a sequentially organized program are illustrated in Figure 2.la. Parallelism is said to exist between $T_{1}$ and $T_{2}$, if the execution of $T_{3}$ is independent of whether task $T_{1}$ and $T_{2}$ are executed sequentially or in parallel as shown in Figure 2.1b.

(a)

(b)

FIGURE 2.1: Sequential and parallel execution of two tasks

Parallelism can exist at several levels within an individual
program. These levels may vary from a statement or group of statements of a procedural language to the level of micro-operations. Program
parallelism or multiprogramming refers to the type of processing in which independent programs are processed concurrently. Intraprogram parallelism, on the other hand, refers to the type of processing in which a single program can be partitioned into tasks that can be performed in parallel i.e. multitasking. Intraprogram parallelism can be classified into global and local. In the global type of parallel processing, a program is partitioned into tasks that can be performed in parallel. Figure 2.2 shows a sample of a FORTRAN program illustrating both global and local parallelism

|  | Read 100 A, B,C,D |
| :---: | :---: |
|  | - ${ }^{\text {a }}$ |
| $\rightarrow X=A * * 2-2.0 * A * B+B * * 2$ |  |
| $\mathrm{Y}=\mathrm{C} * * 2+2.0 * C * D+D * * 2$ |  |
|  | : |
| 10 | $z=(A * B)+(C * D)$ |
|  | - |
|  | CALL SUBl ( $A, B, E$ ) |
|  | CALL SUB2 ( $C, D, F)$ |
| 100 | FORMAT (4EIO.4) |
|  |  |
|  | END |

FIGURE 2.2: Sample Fortran program illustrating global and local parallelism

The two arithmetic expressions for $X$ and $Y$ can be executed in parallel because they each have independent input sets. Parallelism on a local level can be illustrated by statement 10 of Figure 2.2 . As shown in Figure 2.3, "subtasks" (2.3a) and (2.3b) within the task outlined by statement 10 can be executed sequentially or in parallel. A second example of local parallelism, the subroutine (say SUBl) could itself consist of statements of the same form as statement 10 and be executed in parallel themselves.

(a)

(b)

FIGURE 2.3: Illustration of parallelism of a local level

To design a parallel program, it is necessary first to identify the tasks that can be run in parallel. The two approaches which have been used to solve this problem are known as explicit and implicit parallelism. In the explicit approach, the programmer explicitly specifies the concurrency that exists in the program by using additional instructions within the programming language itself. While the implicit approach relieves the programmer of any additional duties, and relies totally upon indicators existing in the program itself.

The two approaches have some advantages and disadvantages. In explicit parallelism, the programmer can change the structure of an algorithm if it is not suitable for parallel processing. While inserting new parallel programming constructs can be a time consuming process and may lead to mistakes. In the implicit approach, it is
independent of the programmer, and existing programs would not have to be modified to take advantage of any inherent parallelism. However, this approach is associated with compiler and supervisory programs to detect the parallelism and their related running overheads.

### 2.2.1 Explicit Parallelism

In explicit parallelism, the users must be provided with programming abstractions that permit them to indicate explicit parallelism when desired in a program. Conway (1963) used FORK and JOIN statements as an approach to parallelism. FORK is an instruction that indicates the initiation of parallel tasks, JOIN waits for a previously created process to terminate. The three ways that FORK may be specified are:

1. FORK A, the execution of this statement initiates another process at address $A$ and continues the current process.
2. FORK $A, J$, the execution of this statement causes the same action as FORK A and also increments a counter at address $J$.
3. FORK $A, J, N$, the execution of this statement causes the same action as JOIN $A$ and sets the counter at address $J$ to $N$.

JOIN $J$ is used with all the forms of the FORK command usage. The execution of JOIN $J$ decrements the counter $J$ by one. The process at address $J+1$ is initiated if the result of the counter at $J$ is equal to zero, otherwise the process executing the JOIN is released. The FORK and JOIN statements in parallel programming are similar in concept to the GOTO statement in sequential programming. An example of the FORKJOIN instructions is illustrated in Figure 2.4.


FIGURE 2.4: FORK and JOIN (a) Flow chart, (b) Instructions

The FORK at location 100 means: set the contents of location 299 to 2 ; then instructions at 200 and at 101 will be subsequently executed. The "2" in the instruction specifies the number of processors that the FORK at location 100 will activate (if they are available). Each processor, when it comes to the end of its parallel path, it branches to location 299 where it decrements the counter by one, and tests if the result is equal to zero. This means it is the last task to finish.

In the example, there are two JOIN instructions, at 106 and 220 . Each one reads: JOIN 299. This means, "decrement the counter 299 by one. If the result is zero, branch to $299+1$, otherwise release this processor ". Notice that FORK has an associativity property; $N$ parallel paths may be specified equally well by many possible arrangements of $\mathrm{N}-1$ forks.

Opler [1965], suggested two statements that allow the programmer to indicate the sections of program which are to be executed in parallel. The two statements are DO TOGETHER and HOLD. These two statements are used to overcome the limitation in the procedure-oriented languages (Algol, Fortran, Cobol, etc.) when used to express a problem solution involving parallelism. The DO TOGETHER will create a range of parallel operations and to define two or more parallel paths within this range. The HOLD will terminate the range created by the DO TOGETHER. The statement after the HOLD statement is executed only after all the executable statements in all the paths have been processed. One of the formats of the DO TOGETHER instruction is:

Label 1 DO TOGETHER label 2, labe1 3,..., label n-1 (label n) Label 1 is optional and indicates the beginning of the range. Lable $n$ is required and the tag of the HOLD that terminates the range, label 2 to label $n-1$ are tags of the first statement in each of the $n-2$ paths. While the HOLD format is:

Label HOLD
The label is compulsory and must be referenced by one or more DO TOGETHER's. Each path in the DO TOGETHER must be logically selfcontained. DO TOGETHER's may be nested and may share the same HOLD. Branching into or out of the range of a DO TOGETHER is not permitted.

As an example:

L1 DO TOGETHER L2,L3 (L4)
L2 Statement 21
Statement 22
Statement 23

L3 Statement 31
Statement 32
Statement 33
 $\rceil$ Path 2

L4
HOLD

The block-structured language proposed by Dijkstra (1965) is an equivalent extension of the FORK-JOIN concept. Cobegin-Coend (or Parbegin-parend) constructs used to concurrently execute each process in a set of $n$ processes $S 1, S 2, \ldots, S n$ as:

## begin

SO;
cobegin $\mathrm{Sl} ; \mathrm{S} 2 ; \ldots$; Sn coend $\mathrm{Sn}+1 ;$
end

The cobegin $\operatorname{sl}$; $22 ; \ldots ; \operatorname{sn}$ coend indicates that statements $\mathrm{Sl}, \mathrm{S} 2, \ldots$, Sn can be executed concurrently. When all the statements $\mathrm{s} 1, \mathrm{~s} 2, \ldots$, , $n$ are executed and terminated, the following statement ( $\mathrm{Sn}+\mathrm{l}$ ) in the program is executed. Figure 2.6 shows the precedence graph of the above example.

The processes defined by the concurrent statement are completely independent of one another. Thus, the set of statements $\mathrm{Sl}, \mathrm{S} 2, \ldots, \mathrm{Sn}$ are disjoint processes. This implies that, to change any variable by a process, that variable must be private to that process, but disjoint


## FIGURE 2.6: Precedence graph of a concurrent program

processes can refer to common variables. The compiler should be capable of detecting the disjointness between processes and determine the variables that can be changed by the processes and those that can be referenced only. Concurrent statements can be nested arbitrarily as in the following example, which is illustrated in Figure 2.7.

## begin

SO;
cobeg in
S1;
begin S2; cobegin S3; S4; S5; coend S6; end
S7;
coend
58;
end


FIGURE 2.7: Precedence graph of nested concurrent processes

Within the execution of the loops statements, parallelism can normally be found. Gosden [1966] has implemented PARALLEL FOR statements and it is noted that his implementation is independent of the number of processors available. As an example, consider the matrix computation $C \leftarrow A . B$, where $A$ is an $n \times n$ matrix and $B$ and $C$ are $n \times 1$ column vectors, for very large n. A parallel For (parfor) statement is used to implement the computation of $C$. The parfor statement will generate p independent processes. Assume that p divides n and $\mathrm{n} / \mathrm{p}=\mathrm{s}$ :
parfor $i \notin 1$ until $p$ do
begin
for $j+(i-1) s+1$ until $s . i$ do
begin
$C(j)+0 ;$
for $k+1$ until $n$ do $C(j)+C(j)+A(j, k) . B(k) ;$
end
end
Each process being generated computes the statements between the outermost begin-end constructs for a different value of i. Hence, the computation of each group of $\mathrm{C}(i)$ is done concurrently.

Shared variables should be controlled while being accessed by different processes in a concurrent processing environment. In concurrent processes, the segments of program that are used to enable one processor to modify a shared variable is called a critical section. There exists at most only one process in a critical section at a time (i.e., mutual exclusion). A number of constructs can be included to protect the use of the shared variables. An example is the MUTEXBEGIN and MUTEXEND construction which is used to mutually exclude access to a set of shared variables. LOCK and UNLOCK constructions may be used to protect the shared variables from simultaneous access. LOCK (X1, X2, ...), makes the data variables $X_{1}, x_{2}, \ldots$, and are the exclusive property of the branch issuing the LOCK statement. The UNLOCK ( $\mathrm{X}_{1}, \mathrm{X}_{2}, \ldots$ ) released the previously locked statements. An explicit notation is needed to specify whether a variable is private to a single processor or shared by several processes. A shared variable $v$ of type $T$ is declared as follows:
var v: shared $T$

Concurrent process can only refer to and change a shared variable inside a critical section. Then critical section may be defined by csect $v$ do $S$
where $S$ is associated with a common variable $v$ and implies that the statement $s$ should have exclusive access to v .

### 2.2.2 Implicit Parallelism

An alternative approach to explicit parallelism is implicit parallelism, where the independent processes are automatically detected by the careful and detailed analysis of the source program. In this approach the compiler scans the source program and detects any independence between statements or program segments. This is considered to be the first step in any parallelization technique. Detection of the relationships between these parts allows the program to be run on a parallel computer. This approach to parallelism is independent of the programmer, i.e. the programmer need not express the parallelism in the problem and a sequential program need not be rewritten to run efficiently on a parallel computer. On the other hand, the compiling and supervisory programs are complex and incur an overhead.

An area in which implicit parallelism can be applied is the detection of parallelism within an arithmetic expression. Gonzalez and Ramamoorthy [1970] and williams [1978] studied the detection of parallelism within arithmetic expressions which are executed on parallel computers with a number of arithmetic units or processors. 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 sequential computer the time taken to calculate an expression can be estimated to be proportional to the number of operations needed to be performed.

If number of available arithmetic units or processors are sufficient in any parallel computer, then any operations that appear at the same level in a tree representation of an arithmetic expression can be executed in parallel on separate processors. As an example, consider the expression,

## $A+B+C+D+E+F+G+H$

which is shown in Figure $2.8 \mathrm{a}, \mathrm{b}$. It is clear that the execution of this expression will require 7 units of time for the tree representation in Figure 2.8 a and 3 units of time for the tree representation in Figure 2.8b. This is because the two representations have a tree of height levels 7 and 3 respectively. On the other hand, the number of processors required for each level will be different in both representations. In Figure 2.8 a , only one processor is required in every level. Whereas, in Figure 2.8 b , four processors are required at level 1 , two at level 2 and one at level 3. So the tree representation of Figure 2.8 b is more suitable for parallel execution than that of Figure 2.8 a , and we can conclude that 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.

Hellerman [1966] has suggested an algorithm that is based on the input expression being presented in reverse Polish notation and containing only binary operators. The input string is scanned from left to right replacing by temporary results each occurrence of adjacent operands immediately followed by one operation. It is a fast and simple


FIGURE 2.8: $\begin{aligned} & \text { Possible binary tree representations of the expression } \\ & \\ & A+B+C+D+E+F+G+H\end{aligned}$ level


FIGURE 2.9: Parallel computation of $A+B * C+D * E * F * G+H+I$ using Hellerman's algorithm
algorithm, but it requires the input in reverse Polish notation and handles commutative operators. An example using this algorithm to handle the expression

## $A+B * C+D * E * F * G+H+I$

is shown in Figure 2.9.

It can be seen that one processor is used at level 1,4,5 and 6 and two processors at levels 2 and 3.
$\not$ Stone [1967] proposed an algorithm based on two subtrees of the same level combined into one a level higher. The algorithm generates its output in a single pass and in reverse Polish notation. The algorithm is slow because of the recursiveness and additional passes which are required to specify parallel computation. Figure 2.10 shows the output obtained from Stone's algorithm applied to the expression: $A+B * C+D * E * F *+H+I$


FIGURE 2.10: Parallel computation of $A+B * C+D{ }^{*} E * F * G+H+I$ using Stone's algorithm

All levels require two processors only while level 4 and 5 require only one processor.

Squire [1963] in his algorithm forms quintuples of temporary results of the form:
$R_{i}$ (operand 1 , operator, operand 2 , start level, end level)
start level $=\max ($ end level operand 1; end level operand 2)
end level $=$ start level+l.
Initially all the variables have a start and end level equal to zero. All temporary results which have the same start level can be computed in parallel. The algorithm scanning the input from right to the left, starts from the right most operator and proceeds until an operator is found with lower priority than that of the previously scanned operator. Now a left to right scan proceeds until an operator is found whose priority is lower than that of the left-most operator of the substring. At this point a temporary result is used. The temporary result replaces one of the operands, and the other is deleted together with its left operator. The left to right scans are repeated until no further quintuple can be produced, and at that time the right to left scan is reinitiated. The algorithm can also handle subtraction and division with increased complexity. Polish notation plays no part in either input or output manipulation. Figure 2.11 shows a parallel computation of the expression:
$A+B * C+D * E * * G+H+I$

Bear and Bovet [1968] in their algorithm use multiple passes over the input string and each pass corresponds to a level. The output string of one level becomes the input string for the next level until the whole expression has been compiled. Thus, the number of passes will


FIGURE 2.11: Parallel computation of $A+B * C+D * E * F * G+H+I$ using Squire's algorithm
be equal to the number of levels in the syntactic tree. A left to right scan is used so that the same symbol is scanned once during each pass. All operations which have the same level number can be performed in parallel. Figure 2.12 shows the syntactic tree generated by this algorithm for the expression
$A+B * C+D * E F * G+H+I$
level


FIGURE 2.12: Parallel computation of $A+B * C+D * E * F * G+H+I$ using Baer's and Bovet's algorithm

## Different approaches have been developed for recognizing

parallelism in a program automatically. Bernstein [1966] in his method suggested conditions which must be satisfied before a sequential process can be executed in parallel which is based on two separate sets of variables for each process $P_{i}$ :

1. The read set $R_{i}$ represents the set of all memory locations for which the first operation in $P_{i}$ involving them is fetch.
2. The write set $W_{i}$ represents the set of all locations that are stored into in $P_{i}$.

Two sequential processes $P_{1}$ and $P_{2}$ can be executed in parallel if they satisfy the following conditions:

1. Locations in $R_{1}$ must not be destroyed by storing operations in $W_{2}$. The areas of memory for which tash $P_{i}$ is read and onto which task $P_{2}$ writes should be mutually exclusive, that is,

$$
R_{1} \cap W_{2}=\phi \quad(\phi=\text { empty set })
$$

2. By symmetry, exchanging the roles of $P_{1}$ and $P_{2}$,

$$
R_{2} \cap W_{1}=\phi
$$

In addition, to maintain the state of the machine when entering $P_{3}$ independently of the mode of execution of $P_{1}$ and $P_{2}, P_{3}$ must be independent of the storing operations in $P_{1}$ and $P_{2}$, that is,

$$
\left(W_{1} \cap W_{2}\right) \cap R_{3}=\phi
$$

Following this work, Evans and Williams [1978] have presented a method of locating parallelism within ALGOL-type programming languages and they investigated some constructs such as the following: assignment statements, loops and IF statements.

To conclude this section, with the increasing complexity of the
problems to be solved and future computer system structures, the
difficulties of program construction has grown. One way to relieve
the difficulties is to make parallel programming more automatic which
will increase efficiency.

### 2.3 PARALLEL LANGUAGES FOR VECTOR PROCESSING <br> In this section we describe the parallel languages features for vector processing that have been developed for existing pipeline computers.

Vectorization is known as the process to replace a block of sequential code by vector instructions. The system software which does the regeneration of parallelism is known as a vectorizing compiler. An intelligent compiler must be developed to detect the concurrency among vector instructions which can be realized with pipelining or with the chaining of pipelines.

High-level languages with parallel constructs have been developed to facilitate vector processing. As we know the use of sequential languages will lose the parallelism specified in a good algorithm. Thus, vectorization is required to restore the concurrency in parallel algorithms so that they can be efficiently implemented on a vector processor. Most commercial vector processors have built-in hardware to support extended high-level languages, like the extended FORTRAN on Cray-1, and the FORTRAN 77 extension in the Fujitsu FACOM VP-200.

Two vector processing languages have been proposed recently: i.e., the Actus by Perrott [1979] and the other is the Vectran by Paul and Wilson [1975]. Neither Actus nor Vectran has been successfully tested on a real machine. Hwang and Briggs [1984] state that the language for vector processing should have the following features:

1. Flexibility in declaring and selecting array elements in the rows, columns, blocks, diagonals and in various subarray expressions.
2. Effectiveness in manipulating sparse and dense matrices.
3. Array conformity to allow transportability.
4. A mechanism to break vectorization barriers.

The following Fortran extension examples show some of the features in a typical parallel language. The extended notation may be specified through an implied DO notation as follows:

```
el:e2:e3
el:e2
*
el:*:e3
```

where el, e2 and e3 are expressions of the indexing parameters as they appear in a DO statement. Only the single symbol "*" indicates that all of the elements are in a particular dimension. The notation "-*" may be used to indicate that the elements are in reverse order.

Example 2.1 Given: DIMENSION X(12), $Y(12,5)$
Then: $X(3: 12: 3)$ represents the elements $X(3), X(6), X(9), X(12)$
$Y(4: 6,5)$ represents the elements $Y(4,5), Y(5,5), Y(6,5)$
$Y(*, 4)$ represents the fourth column of the matrix $Y$
$Y(7,3: *)$ represents the elements $Y(7,3), Y(7,4), Y(7,5)$
Portions of an array should be allowed to be identified using separate names explicitly in a vector statement. No extra storage is allocated for an identified vector, simply because it is a virtual name for a collection of elements in the original vector.

Example 2.2 Given: Real $X(8,8)$
Then: VECTOR $X$ ROW $3(1: 8)$ is a vector consisting of the third row of $x$. VECTOR X DIAG ( $1: 8$ ) represents the diagonal elements of $X$.

VECTOR $X \operatorname{COL} 3(1: 8: 2)$ is the vector consisting of $\mathrm{X}(1: 3)$, $x(3: 3), x(5: 3), x(7: 3)$.

A WHERE statement may allow the programmer to indicate the assignment statements to be executed under the control of a logical array. PACK and UNPACK operations demonstrate the use of control vectors.

Example 2.3 Given: DIMENSION $A(6), B(6), C(B) ;$ DATA $A /-3,-2,1,3,-2,5 /$ Then: PACK WHERE (A.GT.O) B=C causes the elements of $C$ in positions corresponding to "true" in A.gT. 0 to be assigned to the $B$ elements such that $B(1)=C(3), B(2)=C(4), B(3)=C(6)$; UNPACK WHERE (A.GT.O) A=B inserts the elements of $B$ into $A$ in positions indicated by A.GT.O. Thus, $A(3)=B(1), A(4)=B(2)$, $A(6)=B(3)$.

With each element of a vector operand, a basic function may be needed to be computed. For example $A(1: 5)=\operatorname{Cos}(B(1: 5))$ is a vector basic function. Several special vector instructions are shown in the following example.

Example 2.4 Given: DIMENSION $A(30), B(30), C(30)$
Then: $\quad C(4: 11)=\operatorname{VADD}(A(4: 11), B(3: 10))$ performs the vector addition;
$S=\$ \operatorname{IZE}(A(1: 30: 4))$ equals the length of the sparse vector
A(1:30:4);
$S=\operatorname{DOTPD}(A, B)$ forms the dot product of vector $A$ and $B$;
$S=\operatorname{MAXVAL}(A)$ finds the largest value of vector $A$.
A vectorizer is needed to detect parallelism in a sequentially coded program. For a program written in Fortran, a Fortran vectorizer will recognize Fortran constructs that can be executed in parallel.

Precedence analysis and code generation are the two basic phases performed by the vectorizer program. In the analysis phase, the vectorizer performs an analysis of data dependencies and determines the possibility of translating Fortran instruction sequences into a vector syntax.

The following examples illustrate how the conventional Fortran statements are converted into vectorized codes, probably by a vectorizing compiler.

Example 2.5: A simple DO loop containing independent instructions can be converted into a single vector instruction.

```
        DO 10 I=4,60,2
    10 A(I) =B(I+3)+C(I+1)
```

are being converted into a single vector statement:

$$
A(4: 60: 2)=B(7: 63: 2)+C(5: 61: 2)
$$

Example 2.6: A recurrence computation can be converted into vector form, subject only to its precedence constraint. The recursion

$$
A(0)=x
$$

DO $10 \mathrm{I}=1, \mathrm{~N}$
$10 A(I)=A(I-1) * B(I)+C(I+1)$
can be converted to be:

$$
\begin{aligned}
& A(0)=X \\
& A(1: N)=A(0: N: 1) * B(1: N)+C(2: N+1)
\end{aligned}
$$

Example 2.7: An IF statement in a loop can be eliminated by setting a corresponding control vector together with a WHERE statement, such as converting

```
    DO 10 I=1,N
    10 IF(L(I).NE.O)A(I)=A(I)-1
to WHERE (L(I).NE.O) A(1:N)=A(1:N)-1
```

Example 2.8: Parallel computations are sometimes allowed by interchanging the execution sequence, such as converting

DO $10 \mathrm{I}=1, \mathrm{~N}$ $A(I)=B(I-1)$
$10 \quad B(I)=2 * B(I)$
to the following code:
$B(1: N)=2 * B(1: N)$
$A(1: N)=B(0: N-1)$
The loop imposes an ordering which when you unroll the loop can only be done correctly by reordering the instructions.

Example 2.9: Temporary storage can be used to enable parallel computations, such as converting the standards

DO $10 \mathrm{I}=1, \mathrm{~N}$
$A(I)=B(I)+C(I)$
$10 B(I)=2 * A(I+1)$
to vector code:
$\operatorname{TEMP}(1: N)=A(2: N+1)$
$A(1: N)=B(1: N)+C(1: N)$
$B(1: N)=2 * T E M P(1: N)$

A vectorizer informs the programmer of the possibility of parallel operations. The programer can rearrange the computations for better pipelining by examining the output of the vectorizer. Automatic vectorization and code optimization will increase the programming productivity of vector processing.

### 2.4 ARRAY PROCESSING LANGUAGES AND PROGRAMMING

Array processors are also known as SIMD computers since it can handle single instruction and multiple data streams (see Chapter 1). Parallel computation on vector or matrix type of data was certainly the original purpose for developing SIMD array processors. All the processing elements perform the same function synchronously in a lockstep fashion under the command of a control unit. Parallel execution in the array of processing elements is started after all vector operands are stored in the processing elements memory. SIMD array processors allow explicit expression of parallelism in user programs. The compiler detects the parallelism and generates object code suitable for execution in the multiple processing elements and control unit while the control unit is used to execute non-parallel program segments, while parallel executable segments are sent to the processing elements and are executed synchronously. To enable synchronous manipulation in the processing elements, the data is permuted and arranged in vector form. Thus, to run a program more efficiently on an array processor, one must develop a technique for vectorizing the program codes. Parallel programming in SIMD array processors can be seen from the vector operations such as in matrix multiplication.

An example of a SIMD array processor is the Illiac-IV as shown in Chapter l, which was primarily designed for solving partial differential equations and matrix manipulation. Glypnir (Lawrie, et al [1975]), Tranquil (Abel et al (1969]) and Illiac-IV Fortran are three suggested high-level languages for Illiac-IV. Both Glypnir and

Tranquil are Algol-like languages. Tranquil is the first language proposed for Illiac-IV. It has the facilities to manipulate arrays of data in a parallel fashion, independent of the machine organization. Glypnir is a block-structured language and was written to exploit the parallelism in the Illiac-IV architecture.

In Illiac-IV, arithmetic operations are carried out under the control of a mask pattern, each processing element associated with it is a 64 bit (true-false) boolean vector which is used for mask purposes. A processing element is activated when its corresponding bit mask is true and a result of an operation may be delivered.

Consider the Glypnir expression:

$$
A:=B * C
$$

when $A, B$ and $C$ are vectors, each may have up to 64 elements. The above multiplication means that each component of $B$ is multiplied by the corresponding element of $C$ and the resulting product vector is stored in A. However, when $C$ is a scalar, the multiplication will be repeated 64 times in an invisible processing element variable.

Extra facilities are provided such as the rotation and shifting of rows to the left and the right. For and if statements are also provided, but it gives unconventional results. For example, if $A, B$ and $C$ are vectors, the statement

If $A>B$ then $C:=A$ else $C:=B$
will deliver the maximum element of $A$ and $B$ to $C$ and any result in both the then and else statement to be executed. In Glypnir, the programmer is responsible for storage allocation and be constrained to only Illiac-IV rows ( 64 components) or vectors of rows. Illiac-IV

Fortran is used to overcome these constraints, since it allows the user to program with vectors of any length. Extra constructs have been added to the language to allow the shifting and rotation of vectors and array rows. The DO statement have been extended to allow the parallel execution of arithmetic expressions and binary data type can be used to specify bit-control vectors for masking purposes.

Most parallel computers use extensions of existing languages, such as extended Fortran for the Star-100, the CFT language for the Cray-I and the Glypnir language for the Illiac-IV. The language $S L-1$ attempted to bring some of the benefits of structured programming to the Star-100 system. The Vectran language has been developed by the IBM research group to facilitate the application of vector-array processing algorithms. Perrot [1979] introduced another parallelprogramming language for array processors which offers a theoretical extension of the language Pascal and is called Actus.

The Actus language attempts to redress the technology imbalance between the advanced architecture of parallel machines and the development of high level languages for such machines. It is aimed at exploiting parallelism and incorporating some software engineering approaches. The syntax of the language enables the expression of parallelism in a manner which is suited to the problem and which can easily be exploited by a parallel architecture. The main features in the Actus language are described below.

An array is declared in Actus by indicating the maximum extent of parallelism. The syntax can support any number of dimensions. For example, a scalar array is declared as:
var scalar: array[l..m.1..n] of real;
The maximum extent of parallelism is introduced by replacing only one pair of sequential dots ".." by a parallel pair":" as shown,
var parallel: array[1:m,1..n] of real;
This declaration indicates that the array "parallel" of $m \times n$ real numbers for which the maximum extent of the parallelism is $m$. The array "parallel" can be manipulated for $m$ elements at a time since it has been declared as a parallel variable with that extent of parallelism. The extent of parallelism is a central concept to this approach. It is defined for an array processor as the number of processing elements that can logically compute upon a particular data structure at the same time.

Index set can be used by the programmer to identify a particular element of a data type that can be accessed in parallel. An index set is defined with the data declaration,
index index=i:j;
where $i$ and $j$ are constant integer values such that $i \leqslant j$. The elements $i$ to $j$ inclusive will be accessed whenever the index-identifier index is used as a parallel-array index. For example, with the declarations var parallel: array[1:m,l..n] of real;
index interior $=2: m-1$;
interior can be used as the first index of the array "parallel" to access column elements other than the boundary elements. By using the index set the expression becomes more readable and the execution efficiency of the program can be improved.

Shift and Rotate are two alignment operators which are available to enable the movement of data between elements of the same or
different parallel variables. The shift operator causes the movement of the data within the range of the declared extent of parallelism. While the rotate operator, which causes the data to be shifted circularly with respect to the extent of parallelism.

To construct algorithms for parallel machines many of the program constructions which are required in a sequential environment such as assignment, selection, iteration and subprograms are necessary. The essential difference is that in this new environment such manipulations must be performed in parallel. A single extent of parallelism can be associated with each simple or structured statement of the language which refers to one or more than one parallel variable; this extent must be less than or equal to the declared extent of parallelism for the parallel variable involved. For example, the following are valid assignment statements involving parallel variables:

$$
\begin{aligned}
& \text { aa }[1: 100, j]:=\operatorname{aa}[1: 100, j] * b b[i, 1: 100] ; \\
& \text { aa }[10: 90, j]:=\operatorname{aa}[10: 90, j] / b b[i, 10: 90] ;
\end{aligned}
$$

The within construct has been suggested to avoid repeatedly indicating the extent of parallelism for a series of statements (or for a single statement) in which the extent of parallelism will not change. This, in turn, avoids the repeated evaluation of the same extent of parallelism. The form of within is:
within specifier do statement where the quantity specifier is either an index set identifier or an explicit extent of parallelism which is shown as follows. The symbol "\#" is used to indicate the extent of parallelism. The assignment statements:
$a \mathrm{a}[1: 100, j]:=\mathrm{aa}[1: 100, j] * \mathrm{bb}[\mathrm{i}, 1: 100]$;
aa[10:90,j]:=aa[10:90,j]/bb[i,10:90];
could have been written alternatively as:

$$
\begin{aligned}
& \text { within } 1: 100 \text { do } a a[\#, j]:=a a[\#, j] * b b[i, \#] ; \\
& \text { within } 10: 90 \text { do } a a[\#, j]:=a a[\#, j] / b b[i, \#] ;
\end{aligned}
$$

The if and case statements can be used to indicate a choice of several execution paths. For example,
var $a, b:$ array $[1: 100]$ of integer;
if $a[1: 100]>0$ then statement
To solve this problem the anonymous sharp symbol "\#" is again used in the construction of the statement. For example in
if $a[1: 100]>0$ then $a[\#]:=0$;
The sharp represents that subset of the set 1 to 100 for which the corresponding element of 'a' are greater than zero. The effect of executing the statement is to assign the value zero to these elements.

The while statement is the main means for specifying the repeated parallel processing of the data. It is used when the number of times the statement is to be executed is unknown. For example:
while $a[1: 50]<b[1: 50]$ do $a[\#]:=a[\#]+1$
the "\#" symbol identifies those elements of a which are less than their corresponding element in $b$ on each occasion that the comparison of elements is performed and only those elements of a have their value increased by 1. Execution terminates when all the elements of a are greater than or equal to their corresponding element in $b$.

In the situation where the number of times the repetition to be
performed is known, the for construct has been expanded to allow the inclusion of parallel variables in the control, start, finish and increment positions. It then takes the form
for control:=start by increment to finish do statement Functions and procedures can be constructed using the data declarations and program constructs previously introduced and the Pascal rules for subprograms apply. Thus, local variables cannot alter their extent of parallelism by a function or procedure call. In procedure and function definitions, the formal parameter list can consist of either scalar or parallel variables or both.

By introducing the concept of the extent of parallelism into data declarations and modifying existing Pascal language constructs to accommodate the special demands of a parallel processing environment. The advantage of the Actus language is that a problem's parallel nature is expressed directly in the syntax of the language which, in turn, makes efficient use of the machine's computational resources. The user, therefore, does not have to get involved with the detection mechanism of a compiler or with the underlying architecture of the machine on which the program will be executed.

### 2.5 MIMD MULTIPROCESSOR PROGRAMMING

A parallel program for a multiprocessor consists of two or more interacting processes. In an MIMD multiprocessor system we need an efficient notation for expressing concurrent operations. Processes are concurrent if their executions overlap in time. In a MIMD multiprocessor environment necessary changes in the instruction set of multiprocessor machines is needed and then modifications or extensions included in the high-level languages for programing multiprocessors. Both explicit parallelism (Section 2.2.1) and implicit parallelism (Section 2.2 .2 ) may be used to exploit parallelism in a multiprocessor environment.

In a multiprocessor system, synchronization takes an increased importance as it could create overheads that are too high. System performance could be reduced significantly if the synchronization mechanisms are not efficient and the algorithms that use them are not properly designed. Synchronization primitives are implemented either directly in the hardware, microcode or in software. Cooperating processing in a multiprocessor environment must often communicate and synchronize for the execution of one process can influence the other via communication. Now communication between processes are carried out either by use of shared variables or message passing, and this is done via a synchronization mechanism. A process executes with unpredictable speed and generates actions or events which must be recognized by another cooperating process. The set of constraints on the ordering of these events constitutes the set of synchronization required for the operating processes. The synchronization mechanism
is used to delay the execution of a process in order to satisfy such constraints.

Mutual exclusion and condition synchronization are the two types of synchronization which are employed when using shared variables. Mutual exciusion ensures that a physical or virtual resource is held indivisibly. While conditional synchronization occurs when an attempt to access a shared data by a process is delayed until the shared data object state changes to the desired value as a result of other processes being executed.

Many constructs have been used to implement both mutual exclusion and condition synchronization. An example of those constructs are MUTEXBEGIN, MUTEXEND and the usage of LOCK and UNLOCK operations which are shown in Section 2.2.1.

Difkstra proposed two primitives and indivisible operation $P$ and $V$, which can be used to implement the mutual-exclusion efficiently. These operations can be shared by many processes and operate on a special common variable called a semaphore, which indicates the number of processes attempting to use the critical section:
var s: semaphore
Then the primitive $P(s)$ acts as an MUTEXBEGIN of a critical section. The $V(s)$ primitive is MUTEXEND and records the termination of a critical section.

Semaphores are quite general and can be used to program almost any kind of synchronization. While the difficulties that arise from using the semaphore in parallel algorithms makes the algorithm unstructured and prone to error. For example; the misuse of $P$ or $V$
accidentally on a samaphore can make a disastrous effect, since mutual exclusion would no longer be ensured. Another type of difficulty expected, is when the programmer forgets to include the semaphore in a critical section when using it and an error occurs in execution.

To overcome the difficulties in semaphore, Hoare [1972] and Hansen [1972] proposed Conditional Critical Section (CCS). This is a structured and highly user-oriented tool for specifying communication among concurrent processes. Their use allows direct expression of the fact that a process has to wait until an arbitrary condition on the shared variables holds. The variable $V$ may only be accessed within CCS statement that name V. A CCS statement is of the form,
csect $V$ do await $C: S$
where $C$ is a boolean expression and $S$ is a statement list. A CCS statement delays the executing process until the condition $C$ is true, $S$ is then executed. Other CCS statements that name the same resource cannot interrupt the evaluation of $C$ and the execution of $S$. Thus, $C$ is guaranteed to be true when the execution of $S$ begins.

In a multiprocessor system with a high degree of concurrency, the problem of deadlocks will arise. Deadlocks occur when members of a group of processes which hold resources are blocked indefinitely from access to resources held by other processes within the group.

Solution to the deadlock problem have been classified as prevention, avoidance, detection and recovery techniques. Prevention is the process of constraining system users so that requests leading to a deadlock never occur. The scheduler then allocates resources so that deadlocks will never occur. For dead avoidance, the scheduler controls the

[^0]
### 2.6 DATA FLOW LANGUAGES

To overcome the problem encountered in introducing parallelism into Von Neumann model, the data flow model of computing is introduced. A data flow computation is one in which the operations are executed in an order determined by the data interdependencies and the availability of resources. Two varieties of data flow computation can be distinguished i.e.,

1. Data-Driven Computation, in which operations are executed in an order determined by the availability of input data.
2. Demand-Driven Computation, in which operations are executed in an order determined by the requirement for data.

The data flow approach is often associated with the use of datadriven computation.

Data flow languages are programming notations in which data dependencies are expressed directly by program structure. A data flow program in general may be represented by a directed graph, with nodes used to represent the operations (such as addition, multiplication, subtraction, etc.) and the arcs represent the flow of data, and show the data dependencies. An example of data flow program is shown in Figure 2.13.


FIGURE 2.13: Program to calculate the difference between the sum and product of two numbers.

The nodes represent functional operators connected by data links. Each type of operator has input and output links and specifies a function from data values on input links to data values on output link, as shown in Figure 2.14.

$\mathrm{Ol}=\mathrm{Il}+\mathrm{I} 2$

Add

$\mathrm{Ol}=\mathrm{I} 1 / \mathrm{I} 2$
O2=I1 mod $I 2$
Divide

$\mathrm{Ol}=\neg \mathrm{Il}$

Not

During the execution of programs, the notation of tokens flowing down arcs are used to carry data values. When a node has its data token available upon its input arc(s) it generates an appropriate set of data tokens upon its output $\operatorname{arc}(s)$. This is referred to as "firing". Figure 2.15 shows the steps of the computation of the example shown in Figure 2.13.

(a)

(c) sum and product are calculated


(d) difference is calculated

The set of operations of a data flow language are equivalent to the primitive operations of a sequential language. An operator becomes enabled (ready to execute) when tokens are present on all its input links. The execution of all enabled operators is independent and concurrent. Many data flow operators can execute concurrently if each has its required operands. An enabled data flow operator removes the input values from its input links and computes the output values as functions of the input values. The output values are transmitted onto the output links and the operator returns to the inactive state. Operator execution depends only on information local to the operator; there are no global variables or side effects.

Extra operators are needed in order to provide some method of making run-time data-dependent decisions as to what operators are to be carried out. Figure 2.16 shows the two control operators, the switch and the merge. In the switch operator the input token is placed on the output are selected by the control unit. While in the merge operator, it is the programer's responsibility to ensure that only one input arrives at any one time, the input is then placed on the output. Control operators are used to construct the conditional and loop graph as shown in Figure 2.17.

An alternative representation to the graphical programming notation is the textual programming notation as used in most programming languages. Many single assignment and functional programming languages have been developed by various research groups. Among those languages are, the Irvine Data Flow (ID) language (Arvind, Gostelow and Plouffe


## FIGURE 2.16: The control operators



FIGURE 2.17: Data flow graphs representing typical program constructs
[1978]) and the Value Algorithmic Language (VA) (Ackerman and Dennis [1979]).

The main characteristic of the data flow languages ares

1. It is shown to possess great locality. Assignment to a formal parameter should be within a definite range. As a result, data flow languages are appropriate for block structures.
2. Freedom of side effect. The absence of global or common variables and careful control of the scopes of variables make it possible to avoid side effects (such as in procedures that modify variables in the calling program). Also calling by value which is used in data flow languages will solve the aliasing problem.
3. A single assignment rule offers a method to promote parallelism in a program. The rule is to inhibit the use of the same variable name more than once on the left-hand side of any statement. This will greatly facilitate the detection of parallelism in a program.
4. Data flow programs tend to waste memory space for the increased code length due to the single assignment rule and the excessive copying of data arrays.

### 2.7 PARALLELISM IN HIGH-LEVEL PROGRAMMING LANGUAGES

Most of the high-level languages currently used to program parallel processors are extensions of languages designed many years ago for conventional sequential architectures (such as Fortran). Many new algorithmic languages are equipped with facilities to enable them to handle parallel tasks and concurrency. In this section we are going to investigate the parallelism in three high-level programming languages, these languages are Algol-68, Pascal-plus and Ada.

In Algol-68, parallelism is specified syntactically by a collateral phrase consisting of one or more constituents separated by commas. Collateral elaboration of statements can be performed by means of a construction known as a void-collateral-clause. This takes the form of a sequence of units enclosed by begin and end or '(' and ')' and separated from each other by commas. An example of Void-collateralclause is:

$$
(a:=b,((c l:=d 1 ; b 1:=e l),(c 2:=d 2 ; b 2:=e 2)), p r i n t(b))
$$

The relative order in which the various statements are performed can be shown in Figure 2.18.

Facilities for the coordination of control in collateral tasks have been proposed by Dennis and Van Horn [1966] (LOCK/UNLOCK mechanism) and Dijkstra [1965] (samaphores). Algol-68 is equipped for this with the structure sema, the values of which are integers and two operators: up and down.

The two operators up and down are used to permit synchronization to take place. The down operator can involve a potential delay and hence suspension of the process in which it is contained. The up


FIGURE 2.18: The order in which ( $a:=b,((c l:=d l, b l:=e l),(c 2:=d 2 ; b 2:=e 2))$, print(b)) is executed
operator does not involve a delay but its use can awaken or restart a process that was earlier sent to sleep or suspended as a result of a previous application of down. An import precondition concerning semaphores is that no two processes can simultaneously access the same semaphore in order to perform an up or down operation on it.

Whenever synchronization has to take place there must be some form of parallel processing inthe operation. Synchronization can be performed only between the units of a parallel clause. A Efallel clause is like a Void-collateral-clause except that it msut be preceded by par.

Pascal-plus extend the Pascal programming language with process, monitor and condition constructs required for programming a system of parallel activities. The process construct is used to represent program modules whose execution may proceed in parallel. The syntactic form of a process is:
process name \{formal parameter list\};
\{local declaration\}
begin
\{body\}
end;

## instance

P: name \{actual parameter list\};

It is possible that several processes can share the same data and only one allowed to modify it. The monitor construct is used to control shared data as defined by Hoare [1974] and Hansen [1975]. In Pascal-plus a monitor guarantees that only one process at a time can execute its local code and therefore change its local data.

When a process requires access to a monitor it may be forced to wait implicitly if some other process is currently executing its code. In addition, having entered the monitor the process may find that the
data it requires is not available. Synchronization is achieved in Pascal-plus through an operation on a built-in monitor called condition, which is based on the Hoare condition [1974]. The operations provided by this monitor are:

```
monitor condition;
    type range=0..maxint;
    procedure pwait(priority:range);
    procedure wait;
    procedure signal;
    function length:range;
    function priority:range;
```

end \{condition\}

Associated with each instance of condition is an ordered queue on which process may be suspended. When a process invokes the pwait operation it is placed on this queue according to the specified priority value. signal operation is used to activate a process at the head of the condition queue.

The Ada language provides two forms of modules called packages and tasks with similar properties. A module is a programming unit that consists of a specification and a body. Parallel activities are described by means of tasks. A task is primarily intended to introduce a new control path. A task cannot be a stand-alone unit in an Ada program but it must be declared within another program unit, which is referred to as a 'parent' of the task and a task is said to be 'dependent' on its parent. The outline of a task is shown as:
task TASK-NAME is

The task specification

The body of the task must be declared in the same declaration part, but need not immediately follow the specification.
task body TASK-NAME is
$\left\{\begin{array}{l}\text { The declarative part of the task, all objects } \\ \text { declared here are private to the task and not } \\ \text { visible to the other program units. }\end{array}\right\}$
begin
$\left\{\begin{array}{l}\text { The imperative part of the task, this will } \\ \text { contain statements that accept calls on the } \\ \text { entries declared in the specification }\end{array}\right\}$
accept ENTRY-NAME (<parameterlist>) do ...
end TASK-NAME;

In some cases a task presents no interface to other tasks in which case the specification reduces to just
task TASK-NAME;
As an example, consider a family going shopping. The mother buys the meat, the children buy the salad and the father buys the wine. These activities can be illustrated in parallel as:
procedure HOPPING is
task GET-SALAD;
task body GET-SALAD is
begin
BUY-SALAD;
end GET-SALAD;

## task GET-WINE;

task body GET-WINE is
begin
BUY-WINE;
end GET-WINE;
begin

BUY-MEAT;
end SHOPPING;

In this example, the mother is represented as the main processor and calls BUY-MEAT directly from the procedure SHOPPING. The children and the father are considered as subservient processors. The activation of a task is automatic. In the above example the local tasks become active when the parent unit reaches the begin following the task declaration. Such a task will terminate when it reaches its final end. A task declared in the declarative part of a subprogram, block or task body is said to depend on that unit. It is an important rule that a unit cannot be left until all dependent tasks have terminated. It is important to realise that the main program is itself considered to be called by a hypothetical main task. In the SHOPPING example, the sequence of actions when this main task calls the procedure SHOPPING. First the tasks GET-SALAD and GET-WINE are declared and then when the main task reaches the begin these dependent tasks are set active in parallel with the main task. The dependent tasks call their respective procedures and terminate. Meanwhile the main task calls BUY-MEAT and then reaches the end of SHOPPING. The main task then waits until the
dependent tasks have terminated if they have not already done so. Generally, tasks will interact with each other during their lifetime. In Ada this is done by a mechanism known as a rendezvous. A rendezvous between two tasks occurs as a consequence of one task calling an entry declared in another:
entry ENTRY-NAME;
which is then accepted in the body of the task by the statement accept ENTRY-NAME;

A calling task then sends its signal by the entry call:
CALLED-TASK-NAME.ENTRY-NAME;

The interactions between parallel tasks consists of actions of synchronisation and actions of communications. The entry-accept construct may be used to implement synchronisation. Rendezvous can be used to express problems of mutual exclusion in a form similar to monitors or critical regions. Also, the exchange of data between tasks can also be achieved by entries: similarly to procedures, entries can have formal parameters, and after an entry call provides the actual meaning of the parameters.

An important capability, that of conditional execution is provided in the Ada language through the execution of the select statement. The select statement enables a called task to select one of several alternative entry calls. A select statement combines several accept and delay statements. A simple form of the select statement is shown below:

## select

accept ENTRY-ONE do
end ENTRY-ONE;

오
accept ENTRY-TWO do
end ENTRY-TWO;

으
accept ENTRY-THREE do
end ENTRY-THREE;
end select;

The second feature of a select statement is the possibility of guarding any clause by a boolean condition: a call to the corresponding entry will be accepted, or the corresponding delay executed, only if the guard evaluates as true when the select statement is reached. select

## when CONDITION-1 =>

accept ENTRY-1 do
or
when CONDITION-2 =>
accept ENTRY-2 do
Or
when CONDITION-3 =>
accept ENTRY-3 do
else
alternative actions;
end select;

A select statement may end with an else clause which will be executed if no other clause in the select statement can be executed immediately. A select statement can be used to provide a protected data-area to overcome the problem of the shared data-area.

Finally, a delay statement can be used to control the execution of a parallel process, not only with respect to other processes, but also with the flow of time. The statement: "delay $n$;" (where $n$ is an expression yielding a value of the predefined type TIME) will cause the task that executes it to be blocked for at least $n$ units of time.

### 2.8 PROGRAMMING THE NEPTUNE SYSTEM

In Chapter 1, the hardware configuration and software characteristics of the NEPTUNE MIMD sYstem were described. The parallel programming aspects of this system will be illustrated in this section. Two parallel programming languages that are available on the NEPTUNE system are Pascal and Fortran. The adoption of Fortran is much easier due to firstly, the concept of shared memory is very similar to that of the sequential Fortran 'common' block. Secondly, Fortran IV does not permit recursion and storage is statically allocated at compile time. The parallel fortran language was used in the course of this research, and this is the language to be considered in this section.

Many constructs have been added to sequential Fortran to be extended to handle the parallel processing requirements. The properties of the parallel programs that run on the NEPTUNE system are, firstly, only one parallel path (task) is executed on each processor at a time, while other processors must be informed and locked out of that path. Secondly, when this parallel path has been completed the next one in order can be executed by the same processor. Both the local data of this processor and the shared data can be used.

In a parallel programming environment, the user should have the ability to, create and terminate parallel paths, share data between paths and ensure the consistency of data. The essential constructs that are needed to implement any parallel program on the NEPTUNE system are:
(1) \$USEPAR

This construct must be the first executable parallel statement in
the program. 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 sequential Fortran. It forces the checking at pre-compile time and that the nesting of parallel syntactical constructs is completed within each individual subroutine.
(3) \$STOP

This statement replaces the STOP statement of a sequential Fortran program. It ensures the termination of the program.

Three pairs of constructs are available on the NEPTUNE system to create/terminate paths, they are:
(i) \$DOPAR/\$PAREND

This construct has the following form:
\$DOPAR label $\mathrm{I}=\mathrm{N} 1, \mathrm{~N} 2, \mathrm{~N} 3$
$:$
'code using index I'
$\vdots$
label \$PAREND
In this form the generated and terminated paths with identical
code. It is similar to the Fortran DO loop, and ( $\mathrm{N} 2-\mathrm{N} 1+1$ )/N3
paths will generate with each path having a unique value of the loop index $I$. The indexing of data by the variable $I$ then allows different paths to evaluate different results. \$PAREND is similar to the CONTINUE statement in Fortran, and by executing the \$PAREND every generated path should be terminated.
(ii)
\$FORK/\$JOIN
This construct has the following form:


In this form, the generated paths are with different code, and it is equivalent to the computed GO TO statement in Fortran. The paths start at labels label 1, label 2, label 3 and terminate at label 4. Each path contains a GO TO statement at the end to force the paths to terminate at label 4 of the \$JOIN statement, except the last path which already terminates at label 4. \$DOALL/\$PAREND

This construct has the form:


In this form, the generated paths are with the same code where each processor is forced to execute the code once and only once, and it is normally used to initialise the data or to obtain the timing.

Shared variables may be used as a communication means between the different parallel paths, and can be defined as:
\$SHARED variable list
which enforces the variables to be loaded into the shared memory. Whilst all other data including the program code are held in the local memory. This construct is similar to the COMMON statement in Fortran.

In the NEPTUNE system, critical sections are used to enforce sequential access to certain shared data structures to ensure their integrity. For this purpose the user has available up to 8 'resources' which can only be owned by one of the processors at any one time. Resources used must be declared with Fortran-like names using
\$REGION list of names The scope of this declaration is the next \$END construct. The constructs, \$ENTER and \$EXIT are used to resources which are embedded in a critical section and released with \$EXIT, i.e.,
\$ENTER name
\$EXIT name
i.e., the critical section is embedded within an $\$$ ENTER/\$EXIT pair of constructs. The same resources can protect different critical sections in the program.

Many special commands have been introduced to the compiling, linking and running of parallel programs on the NEPTUNE system. The XPFCL or XPFCLD (used when the number of parallel paths up to ( $2^{15}-1$ )) commands are used to produce a load module from the user's source program. The effect of these two commands are to:
(i) Preprocess the user source program to convert the special parallel constructs to FORTRAN statements. Compile the resultant FORTRAN. Link the compiler output with the available FORTRAN libraries and machine code written routines to control the parallelism. store the resultant load module in the user's program file.

If an error is detected in any one of the above stages, an error message will be written to the output file of that stage and the command is terminated.

To run a parallel program on the NEPTUNE system, the XPFT command is used. The user should specify the processor(s) on which the load module should run, the name of the load module, and whether the execution is required on the foreground or background computing basis. The processors in the system are numbered 0 to 3 and processor 0 must be one of these processors listed because it includes the main disc. Any error during the run time will be reported and the run is terminated.

The Performance measurement is an important issue in parallel programming and timing is an essential factor of its analysis. Two subroutines are available for obtaining timing information. The routines should be embedded within a \$DOALL/\$PAREND sequence to force each processor to execute time, and the timing results should appear in the output file. To force to activate the timing for each processor, the following constructs are used:
\$DOALL 10
CALI TIMEST
\$PAREND

Normally these constructs are placed before the part of the program to be performed by the path. To get the timing results for each processor and place it in an array ITIME (as an example), the following constructs are used:
\$DOALL 20

## CALL TIMOUT(ITIME)

## 20 \$PAREND

These constructs are placed at the end of the part of the program in the path. Usually the ITIME is declared as a shared array of size 100 and this 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 information for processor $j=0,1,2,3$. With $i=j * 25$, then we have: $\left.\begin{array}{l}\text { ITIME (l+i) } \\ \operatorname{ITIME}(2+i)\end{array}\right\}$ clocked cpu time in seconds and milliseconds


ITIME(5+i) number of parallel paths runs by this processor

ITIME (6+i) number of waiting cycles because no path is available

ITIME(7+i) number of accesses to critical section resource 1

ITIME(8+i) number of accesses to critical section resource is being used by another processor


## CHAPTER 3

DESIGN OF PARALLEL ALGORITHMS

### 3.1 INTRODUCTION

Researchers have studied parallel algorithms even before parallel computers had been constructed. Regardless of whether parallel algorithms will be used in practice, many researchers find designing parallel algorithms fascinating and challenging. Designing parallel algorithms became more important and interesting as the development of parallel computer architecture advances. Therefore, different parallel algorithms have been designed from different viewpoints and for the various parallel architectures which were described in Chapter 1. In this section, the classification of parallel algorithms corresponds naturally to that of parallel architectures. Parallel algorithms can be considered as a collection of independent task modules that can be executed in parallel and that they communicate with each other during the execution of the algorithm. By independent task module we mean, that the results obtained from one module are not affected by the results obtained from the other. For example, to find the result of adding two $n$-vectors $A$ and $B$,

$$
C=A+B
$$

where $A=\left(a_{1}, a_{2}, \ldots, a_{n}\right), B=\left(b_{1}, b_{2}, \ldots, b_{n}\right)$ and $C=\left(c_{1}, c_{2}, \ldots, c_{n}\right)$. The evaluation of the components of the resultant vector $C$ are obtained from the formula,

$$
c_{i}=a_{i}+b_{i}, \text { for } i=1,2, \ldots, n
$$

It is clear that the calculation of the components of the vector $C$ are independent and a computer with $n$ processors takes one step to compute the vector $C$, where each component is evaluated in one processor. The example shows that the algorithm already contains independent computations without the need to reorganise, i.e., the example is said
to exhibit inherent parallelism. A parallel algorithm may be created by identifying the inherent parallelism of a sequential algorithm. An algorithm may be reconstructed in order to increase the property of parallelism.

Stone [1973] highlights some of the problem areas in parallel computation, these include the necessity to rearrange the data in memory for efficient parallel computation; the recognition that efficient sequential algorithms are not necessarily efficient on parallel computers and conversely, that sometimes inefficient sequential algorithms can lead to very efficient parallel algorithms and lastly the possibility of applying transformations to sequential algorithms to yield new algorithms suitable for parallel execution.

Kung [1980] identified three orthogonal dimensions of the space of parallel algorithms: concurrency control, module granularity and communication geometry. Concurrency control is needed in parallel algorithms to ensure the correctness of the concurrent execution, because more than one task module can be executed at a time. The module granularity of a parallel algorithm refers to the maximal amount of computation a typical task module can do before having to communicate with other modules. The module granularity of a parallel algorithm reflects whether or not the algorithm tends to be communication intensive. This must be taken into consideration for efficiency reasons. Suppose that the task modules of a parallel algorithm are connected to represent intermodule commanication. Then, a geometric layout of the resulting network is referred to as the communication geometry of the algorithm. It is necessary to take into account the computer into which the parallel algorithm is implemented. As shown in Chapter 1, parallel
computers may be classified into SIMD, MIMD and pipelined computers. In SIMD computers, the number of processors tend to be large compared with that of MIMD computers. In general, we can say that the algorithm designed for SIMD computers requires a high degree of parallelism because this type have up to order $n^{m}$, i.e. $O\left(n^{m}\right)$ processors, while a MIMD computer has up to $O(n)$ processors, where $n$ is the order of the problem (number of subtasks) and $m \geqslant 2$. This does not mean that an algorithm designed for a MIMD computer cannot be run on an SIMD computer, but an algorithm with only $n$ independent computations will be executed concurrently on only $n$ processors and the remaining processors are left idle. Conversely, in an MIMD computer there are insufficient processors to run $O\left(n^{m}\right)$ independent computations concurrently, but instead may execute them in groups of $P$ computations if $P$ processors are available. SIMD computers cannot take advantage of independent computations that are not identical since the processors are synchronous, while the processors of MIMD type are asynchronous and can take advantage of such computations. The non-identical computations could be executed sequentially on the SIMD computer, but the system efficiency would be degraded. However, in the MIMD computer, the processors need not necessarily be involved on the same problem. Thus, when 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, a good MIMD algorithm is not always a good SIMD algorithm and vice versa.

In a pipeline computer, the speed-up is achieved by producing a string of identical operations that may be queued up and treated in an assembly line fashion. It is obvious to see that the string of operations
must be independent and the longer the string the greater the speed-up that is achieved. For this reason, a good pipeline algorithm is generally a good SIMD algorithm and vice versa.

One area in which pipelining appears to be particularly appropriate is in the implementation of arithmetic operations, and it is known as arithmetic pipeline. Arithmetic pipelines have been constructed for performing a single arithmetic function, e.g. floatingpoint addition, or for performing all four basic operations on both fixed-point and floating-point numbers. For example, the Cray Research CRAY-1 uses six-stage floating-point adders and seven-stage floatingpoint adders, and the CDC STAR-100 uses four-stage floating-point adders. For a pipeline floating-point adder, the pipe typically consists of stages for performing exponent alignment, fraction shift, fraction addition and normalization. A pipeline arithmetic unit can be viewed as a systolic machine composed of linearly connected processors that are capable of performing a set of (different) operations.

### 3.2 PARALLEL ALGORITHMS FOR SIMD AND PIPELINE COMPUTERS

As shown in a previous section a good SIMD algorithm is generally a good pipeline algorithm, and vice versa. In this section we will discuss the structure of parallel algorithms for both SIMD and pipeline computers. The research concerning the structure and design of the parallel algorithms for SIMD computers can be found in many papers in the literature, among those are Miranker [1971], Stone [1971],[1973b], Heller [1978] and Wyllie [1979]. While, Chen [1975] and Ramamoorthy and Li [1977] consider the applications of arithmetic operations on pipeline computers.

One of the most successful applications on pipeline computers is the execution of arithmetic operations. The pipeline approach is ideal for situations where the same sequence of operations will be invoked very frequently, so that the start-up time to initialize and fill the pipe become relatively insignificant.

An example of a pipeline adder using a linear array is given in Chen [1975]. To add two $n$-vectors, $\left(U_{1}, U_{2}, \ldots, U_{n}\right)$ and ( $\left.V_{1}, V_{2}, \ldots, V_{n}\right)$ and that $U_{i}=U_{i 1}, U_{i 2}, \ldots, U_{i k}$ and $V_{i}=V_{i 1}, V_{i 2}, \ldots, V_{i k}$ represent their binary representations. Figure 3.1 shows how the adder works for $K=3$. The $U_{i j}$ and $V_{i j}$ march toward the processors synchronously as shown.

At each cycle, each processor sums the three numbers arriving from the three input lines and then outputs the sum and the carry at the output lines. From Figure 3.1, it is easy to check that when the pair $\left(U_{i j}, V_{i j}\right)$ 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. As a result, the pipelined adder can compute a sum $U_{i}+V_{i}$ every cycle in the steady state.


FIGURE 3.1: Pipeline integer adder

This algorithm is suitable for an SIMD computer without considering the binary representations of the digits. As described in Section 3.1, the sum of two $n$-vectors can be computed in one step using $n$ simultaneous processors. The 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 then using,

$$
c_{i}=a_{i}+b_{i}, \text { for } i=1,2, \ldots, n
$$

It is clear that the addition may be performed in one step using ( $\mathrm{n} . \mathrm{m}$ ) processors.

Tang and Lee [1984] suggest a method to design parallel algorithms
for SIMD computers based upon the divide-and-conquer strategy. The divide-and-conquer scheme can be briefly specified as follows: Given a problem and the initial conditions, divide it into $K$ subproblems. once these subproblems have been solved, combine their solutions into a solution for the original problem.

As an example, consider the problem of finding the maximum of a set of $N$ numbers. If the divide-and-conquer approach is used, we divide the set into two subsets $S_{1}$ and $S_{2}$, each subset consisting of $N / 2$ numbers. In each subset $S_{i}$, the algorithm is applied recursively to find its maximum $M_{i}$. The final step is a merging step, which is selecting the maximum from $M_{1}$ and $M_{2}$. Figure 3.2 shows an example where $\mathrm{N}=16$. Each square represents a process which selects the maximum from two numbers.


FIGURE 3.2: A recursive doubling technique to obtain the maximum
of $n$ numbers.


Step2logN+1: One processor finally merges the two solutions into one final solution.

Figure 3.3 shows the above steps.

Tang and Lee show in their analysis, that they can choose the number of processing elements that give the optimal speed-up ratio. They show that, if the number of processing elements used is equal to $N / \log N$, then the complexity of the algorithm with $N / \log N$ processing elements, is of $O(\log N)$ and the speed-up ratio is $O(N / \log N)$, which is optimal.

Another powerful method used for generating parallel algorithms


FIGURE 3.3: Divide and conquer approach
 splitting the data into two parts $\square$ : processing the data with aize one. $\square$ : combining the two solutions into one).
for a SIMD computer is based on problem decomposition is called recursive doubling. The idea is to recursively divide the original computation into independent smaller computations of equal complexity, which are then computed in parallel. As an example, consider the sum of $n$ numbers, $\sum_{i=1}^{n} a_{i}$, then,

$$
S_{n}=\sum_{i=1}^{n} a_{i}=\sum_{i=1}^{m} a_{i}+\sum_{i=m+1}^{n} a_{i} \text { where } m=\lceil n / 2\rceil
$$

by repeated splitting and the sum $S_{n}$ will be evaluated in $\left\lceil\log _{2} n\right\rceil$ steps using $\lceil\mathrm{n} / 2\rceil$ processors, where $\lceil\mathrm{x}\rceil$ is defined as the smallest integer greater than x .

Figure 3.4 illustrates an evaluation tree of the above procedure where n is given the value 8 , and at each level the operations are identical and independent, therefore they can be executed simultaneously.

Actually, any associative operation (such as,,$+ \times,-, \ldots$ ) could be used instead of addition. Heller [1978] named this algorithm the associative fan-in algorithm, but it is also known as the log-sum and log-product algorithms with the operators + and $\times$ respectively. Heller also shows that beside the simplicity of the associative fan-in algorithms, they are optimal in the sense of achieving minimal computation time for any number of processors used.

An example of an optimal algorithm is given below to compute,

$$
A_{n}=a_{1} \circ a_{2} \circ \ldots \circ a_{n},
$$

where $O$ is any associative operation as shown in Figure 3.5. At each level the operations are identical and independent, therefore they can be executed simultaneously. It is clear from the Figure 3.5 that the number of levels is $\log \{n / 2\}$ and by using $\lceil n / 2\rceil$ processors the result $A_{n}$ may be evaluated in $\left\lceil\log _{2} n\right\rceil$ steps.

Level 3

Level 2
Level 2


FIGURE 3.4: The evaluation tree of $\sum_{i=1}^{8} a_{i}$

Level 4

Level 3

Level 2

Level 1


FIGURE 3.5: Evaluation tree of expression $A$

An important application of the associative fan-in algorithm is the computation of inner product (or scalar product). Given two nvectors $A$ and $B$ which have the form,

$$
\begin{array}{r}
s=\quad \sum_{i=1}^{n} a_{i} b_{i}, \quad \text { where } A=\left(a_{1}, a_{2}, \ldots, a_{n}\right) \text { and } \\
B=\left(b_{1}, b_{2}, \ldots, b_{n}\right)
\end{array}
$$

The evaluation tree of the inner-product algorithm is shown in Figure 3.6. The $n$ products are "independent and may be performed simultaneously using $n$ processors, and the sum of the resultant product is performed in $\left\lceil\log _{2} n\right\rceil$ steps. Therefore, the inner product requires $\left\lceil\log _{2} n\right\rceil+1$ steps using $n$ processors.


FIGURE 3.6: Evaluation tree of the inner product algorithm

The inner product algorithm can be extended to evaluate matrix multiplication. More generally, the product of $m \times n$ and $n \times p$ matrices could be optimally computed in $\lceil 10 g n\rceil+1$ steps using (m.n.p) processors, since each component of the m.p results is an inner product of $n$-vectors.

Implementation of algorithms for matrix multiplication on SIMD computers has been discussed by many researchers. Muraoka and Kuck [1973] consider the evaluation of a conformable sequence of matrix products $A_{1}, A_{2}, \ldots, A_{n}$, where $A_{i}$ is either $1 \times N$, $N \times N$ or $N \times 1$ using unlimited parallelism. Hockney and Jesshope [1981], suggest three options by which matrix multiplication may be obtained.

Given three matrices $A, B$ and $C$, where

$$
C_{i j}=\sum_{k=1}^{n} A_{i k} B_{k j}, 1 \leqslant i, j \leqslant n .
$$

Three different options have been suggested to evaluate $C_{i j}$. The first option, the Inner Product Method which is an extension of the inner product algorithm requires $n^{2}\left(\left\lceil\log _{2} n\right\rceil+1\right)$ steps using $n$ processors, since it consists of $n^{2}$ inner products. The second option, the Middle Product Method, which computes the inner product over all the elements of a column of $C$ in parallel and requires $2 n^{2}$ steps using $n$ processors. The parallelism it exhibits is therefore $n / 2$, compared with $\left\lceil\log _{2} n\right\rceil+1$ for the first inner-product option. The third option, the Outer Product Method, which computes the inner-product over all the elements of the array $C$ in parallel requires $2 n$ steps using $n^{2}$ processors. In addition, Jesshope and Craigie [1980] note that the product of the two matrices can be achieved in $\left\lceil\log _{2} n+\bar{\eta}\right.$ steps using $n^{3}$ processors.

In Maruyama [1973] and Kuck and Maruyama [1975] discussed the parallel evaluation of arbitrary matrix expressions with unlimited parallelism.

The design of parallel algorithms may involve the restructuring of the sequential algorithm into a form that is usually a combination of these basic computations. Chen and Kuck [1975], in their algorithm for the solution of a triangular system of operations, is basically a sequence of matrix sums and products.

Unlimited parallelism has been assumed while developing the above mentioned parallel algorithms. This often leads to an algorithm that requires an impractical large number of processors. The original algorithm should be reconstructed into a second algorithm with its processor's requirements reduced to a realistic number. The efficiency of the new algorithm should be the same as using a large number of processors. Two basic principles have been suggested by Hyafil and Kung [1974] in which the new algorithm is constructed, namely the algorithm decomposition and the problem decomposition principles. In the algorithm decomposition principle, it is assumed that $q_{i}$ operations are performed during step $i$ of the original algorithm. In the constructed algorithm $\left\lceil q_{i} / p\right\rceil$ steps are required to perform step $i$, where p is the number of processors available. This means, the decomposition takes place in each step. On the other hand, in the problem decomposition principle 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.

Non-numerical algorithms have also been developed on a SIMD computer. Wyllie [1979] presented algorithms mainly applied to various data structures, such as the counting of the number of elements in a linked list and the deletion of an element from a linked list. Another widely used non-numerical problem is the sorting of a number of keys.

[^1]
### 3.3 PARALLEL ALGORITHMS FOR MIMD COMPUTERS

A parallel algorithm for a multiprocessor is a set of $n$ concurrent processes which may operate simultaneously and cooperatively to solve a given problem. Synchronization and the exchange of data is needed between processes to ensure that the parallel algorithm works correctly and effectively to solve a given problem. Therefore, in some stage in the execution of a process there may be some points where the processes communicate with other processes. These points are called the "interaction points". The interaction points divide a process into stages. Therefore, at the end of each stage, a process may communicate with some other processes before the next stage of the computation is initiated.

Parallel algorithns for multiprocessors may be classified into asynchronous and synchronous parallel algorithms. Because of the interactions between the processes, some processes may be blocked at certain times. The parallel algorithm in which some processes have to wait on other processes is called a synchronized algorithm. The weakness of a synchronized algorithm is that all the processes that have to synchronize at a given point wait for the slowest amongst them. To overcome this problem, an asynchronous algorithm is suggested. In an asynchronous parallel algorithm, processes are not generally required to wait for each other and communication is achieved by using global variables stored in shared memory. Small delays may occur due to concurrent accesses to the common memory.

An algorithm must be decomposed into a set of processes before it can be executed into a multiprocessor system. Hwang and Briggs [1984] describe two decomposition methods, these are the static decomposition
and dynamic decomposition. In a static decomposition, the set of processes and precedence relations are known before execution. In this method, the communication between processes may be very low, provided the number of processes are small; however their adaptability is limited. While in dynamic decomposition, the set of processes changes during execution. In this method, the process communications are high but it can adapt effectively to variations in the execution time of the process graph.

As an example of a synchronous parallel algorithm, suppose it is required to compute the matrix,

$$
Z=A \cdot B+(C+D) \cdot(I+G)
$$

by maximum decomposition. Three processes may be created to compute $z$ synchronously as shown in Figure 3.7. Part of a parallel program that computes the value of $Z$ using three processes is shown below.
var W,Y: shared real; var Sw,Sy: semaphore;
initial $S w=S y=0$;
cobegin
Process Pl: begin
$\mathrm{V} \leftarrow \mathrm{A} \times \mathrm{B}$; //stage 1 of $\mathrm{Pl} / /$
P(Sy);
$\mathrm{Z} \leftarrow \mathrm{V}+\mathrm{Y}$; //stage 2 of $\mathrm{Pl} / /$
end
Process P2: begin
W-C+D; //stage 1 of $\mathrm{P} 2 / /$
V(Sw);
end
Process p3: begin
$\mathrm{X}+\mathrm{I}+\mathrm{G}$; //stage 1 of $\mathrm{P} 3 / /$
P(Sw);
$\mathrm{Y} \leftarrow \mathrm{W}+\mathrm{X}$; $\quad$ stage 2 of P 3
end
coend

From Figure 3.7, it is clear that the set of processes P1,P2 and P3 is a synchronized parallel algorithm because some stages are not activated unless the other processes are completed. The second stage of process P3 is not activated unless the process P2 is completed. Similarly, the second stage of P1 cannot be initiated unless the second stage of P3 is completed. Hence, the set of processes P1, P2 and P3 is a synchronized parallel algorithm.


FIGURE 3.7: Example of a synchronized algorithm with synchronizing stages $(Z=(A . B)+(C+D) .(I+G))$

Now if we consider another example which evaluates the sum of $n$ numbers $\left(\sum_{i=1}^{n} a_{i}\right)$ that is mentioned in Section 3.2 , is:

$$
s=a_{1}+a_{2}+\cdots+a_{n}
$$

The associative fan-in algorithm that is specified in Section 3.2 will be used to evaluate $S$. We assume that the MIMD computer onto which the algorithm to be implemented has $P$ processors. Therefore, the
expression $S$ can be partitioned into $P$ subsets. Thus,

$$
s=\left(a_{1}+\ldots+a_{\ell}\right)+\left(a_{\ell+1}+\ldots+a_{2 \ell}\right)+\ldots+\left(a_{(p-1) \ell}+\ldots+a_{p \ell}\right)
$$

where $\ell=n / p$ is a multiple of $p$ and a parallel program for the evaluation of the expression $S$ using the NEPTUNE programming constructs is shown in Figure 3.8.

It is clear in this program the $P$ paths (processes) will run concurrently. At the end of each path $I$, where $I=1,2, \ldots, P, X(I)$ contains the partial results of the computation of the subexpression which corresponds to that which is evaluated by path I. A critical section is used to enforce sequential access to variable $s$. When all paths have been completed, $S$ will contain the total value of the whole expression.
\$SHARED S
\$REGION REGI

$\mathrm{S}=0.0$
NPATH $=\mathrm{N} / \mathrm{P}$
\$DOPAR 10 I=1,P
$\mathrm{X}(\mathrm{I})=0.0$
IBEGIN $=$ NPATH* $(\mathrm{I}-1)+1$
IEND=NPATH*I
DO $20 \mathrm{~K}=$ IBEGIN, IEND
$\mathrm{X}(\mathrm{I})=\mathrm{X}(\mathrm{I})+\mathrm{A}(\mathrm{K})$
20 CONTINUE
\$ENTER REG1
$\mathrm{S}=\mathrm{S}+\mathrm{X}(\mathrm{I})$
\$EXIT REG1
10 \$parend
FIGURE 3.8: Parallel evaluation of $\sum_{i=1}^{n} a_{i}$
In asynchronous parallel algorithms, there is no explicit dependency
between processes as in synchronized parallel algorithms and global variables or shared data is used as a communication means between processes. Also in asynchronous parallel algorithms the processes never wait for inputs at any time but continue execution or terminate according to the current information in global variables. However, processes may be blocked from entering a critical section which are needed in many applications.

Kung [1976] shows an example of asynchronous parallel algorithms that may be illustrated using the iterative method. For example, the zeros of function $f$ may be computed by Newton's iterative method;

$$
x_{i+1}=x_{i}-f^{\prime}\left(x_{i}\right)^{-1} f\left(x_{i}\right)
$$

where $f^{\prime}(x)$ is the derivative of $f(x)$. Figure 3.9 shows Newton's iterative program using two processors P1 and P2. In the program three global variables $v_{1}, v_{2}$ and $v_{3}$ have been used to hold the current values of $f(x)$, $f^{\prime}(x)$ and $x$ respectively. Suppose the evaluation of $f^{\prime}(x)$ is computationally more expensive than that of $f(x)$, then an asynchronous iterative algorithm consisting of two processes Pl and P2 can be defined as follows. Let process $P 1$ update variables $v_{1}$ and $v_{3}$, while process P2 updates $v_{2}$ as shown on the next page.

It is seen from the program that, as soon as a process completes updating a global variable, it proceeds to the next updating by using the current values of the relevant variables without any delay. If the initial values of the variables are $v_{1}=f\left(x_{0}\right), v_{2}=f^{\prime}\left(x_{0}\right)$ and $v_{3}=x_{1}$, then the timing diagram in Figure 3.10 illustrates the sequence and time period of a step completion for each iteration within each process.

```
function f,f';
var V1,V2,V3: shared real;
cobegin
Process Pl: begin
                            while < termination criteria S not satisfied > do
                            begin
                    V1&f(V3); //step 1 of P1//
                    V 3 +V3-v-1 v
                    end
                    end Pl
                    Process P2: begin
                    while < termination criteria S not satisfied > do
                    V2&f'(V3); //step 1 of P2//
                    end P2
coend
```

FIGURE 3.9: Newton's iterative parallel program


FIGURE 3.10: Time dia'gram for an asynchronous parallel algorithm

The number 1 in the circle on the timing diagram indicates the point where the ith iteration starts for that process. Then,

$$
\begin{aligned}
& x_{2}=x_{1}-f^{\prime}\left(x_{0}\right)^{-1} f\left(x_{1}\right) \\
& x_{3}=x_{2}-f^{\prime}\left(x_{1}\right)^{-1} f\left(x_{2}\right) \\
& x_{4}=x_{3}-f^{\prime}\left(x_{2}\right)^{-1} f\left(x_{3}\right)
\end{aligned}
$$

From the concurrent program given above for P1 and P2, the recurrence relation that is generally followed by the execution of the process is

$$
x_{i+1}=x_{i}-f^{\prime}\left(x_{j}\right) f\left(x_{i}\right)
$$

where $j \leqslant i$. Therefore, the iterates generated by the asynchronous iterative algorithm are different from those generated by the sequential algorithm or synchronized iterative algorithms.

From the algorithm discussed earlier, we notice the difference in the implementation on SIMD and MIMD computers. This follows from the difference in the characteristics of the two types of computers. It is known that the processors of a SIMD computer are synchronized as well as being synchronous, i.e., each process executes the same instruction, 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]).

Parallel algorithms for multiprocessors are studied by Kung [1976], [1980] including both synchronized and asynchronous algorithms. Baudet [1976], Barlow et al [1982] studied the performance analysis of algorithms on asynchronous parallel processors. Iterative techniques
for solving linear systems of equations are given in Conrad and Wallach [1977]. Barlow and Evans [1984] developed parallel algorithms for the iterative solution to linear systems. Evans and Yousif [1985], Yousif and Evans [1985a,b] have studied different implementations of sorting and merging algorithms on the MIMD computer.

### 3.4 VLSI AND SYSTOLIC ALGORITHMS

The development in hardware technology has had a great effect on computer design. It is now feasible to build low-cost, special purpose, peripheral devices to rapidly solve sophisticated problems. VLSI offers outstanding opportunities for inexpensive implementations of highperformance devices. It is clear in VLSI technology that simple and regular interconnections lead to cheap implementations and high densities, and high density implies both high performance and low overhead for support components (Mead and Conway, [1980]). For these reasons, to design an effective parallel algorithm for VLSI processors there has to be a simple and regular data flow. To minimize execution time, pipeline techniques may be used, i.e., processing may proceed concurrently with input and output. The best performance may be obtained by using pipelines and multiprocessing at each stage of processing.

As long as the communication in VLSI remains restrictive, locallyinterconnected arrays will be of great importance. An increase of efficiency can be expected if the algorithm arranges for a balanced distribution of work load while observing the requirement of locality, i.e., short communication paths. These properties of load distribution and information flow serves as a guideline to the designers of VLSI algorithms, and may eventually lead to new designs of architecture and language. Systolic and wavefront arrays are the first such specialpurpose VLSI architectures, which boast tremendously massive concurrency. Kung [1985] shows that to map an algorithm onto an array, a notation should be used so it can be easily understood and compiled into an efficient VLSI array processor. Thus, a powerful expression of array algorithms will be essential to the design of arrays. Systolic and
wavefront expressions are two ways of array algorithm formation.

As shown in Chapter 1, a systolic array is very tractable to VLSI implementation. A systolic system is a network of processors which compute and pass data through the system. In a systolic system, every processor regularly pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network.

Basic processing cells used in the construction of systolic arithmetic arrays are the additive multiply cells specified in Figure 3.11. This cell has the three inputs $A, B, C$ and the three outputs $A=A$, $B=B$ and $D=C+A * B$ (inner product). Six interface registers are assumed to exist and are attached to the input/output ports of a processing cell. All registers are clocked for synchronous transfer of data amongst adjacent cells. The additive-multiply operation is needed in performing the inner product of two vectors, matrix-matrix multiplication, matrix inversion and L-U decomposition of a dense matrix. The processor is capable of performing the inner product step and is called the inner product step processor.


A


C

FIGURE 3.11: Geometries for the inner product step processor

Many different array connections are suggested for different compute-bound algorithms. An example of connected processor arrays are the linearly, the orthogonally and the hexagonally mesh-connected (or hex-connected) scheme as shown in Figure 3.12.

(a) Linearly connected


(b) Orthogonally connected
(c) Hexagonally connected

FIGURE 3.12: Mesh-connected processor arrays

Consider, for example, the construction of a systolic array for the matrix-vector multiplication which is shown by Mead and Conway [1980]. Given a matrix $A=\left(a_{i j}\right)$, a vector $x=\left(x_{1}, x_{2}, \ldots, x_{n}\right)^{T}$ and a vector $y=\left(y_{1}, y_{2}, \ldots, y_{n}\right)^{T}$ which can be computed by the following recurrences:

$$
\begin{aligned}
& y_{i}^{(1)}=0 \\
& y_{i}^{(k+1)}=y_{i}^{(k)}+a_{i k} x_{k} \\
& y_{i}=y_{i}^{(n+1)}
\end{aligned}
$$

Suppose A is an $n \times n$ band matrix* with width* $w=p+q-1$. Then the above

[^2]recurrence can be evaluated by pipelining the $x_{i}$ and $y_{i}$ through w linearly connected processors. Figure 3.13 illustrates the algorithm for the band matrix-vector multiplication problem. While Figure 3.14 shows the linearly connected network for the matrix-vector multiplication problem shown in Figure 3.13.

In general, the matrix-vector multiplication algorithm is carried out as follows:

Initially $y_{i}$ is initiated to zero and moves to the left, while the $x_{i}$ moves to the right and the $a_{i j}$ are moving down. All the moves are synchronized. Each $y_{i}$ is able to accumulate all its terms, namely, $a_{i, j-2} x_{i-2}, a_{i, j-1} x_{i-1}, a_{i, i} x_{i}$ and $a_{i, i+1} x_{i+1}$, before it leaves the network.


FIGURE 3.13: Multiplication of $a$ vector by a band matrix with $p=2$ and $q=3$

The first seven steps of the algorithm can be illustrated in Figure 3.15.

Note that when $Y_{1}$ and $y_{2}$ are output they have the correct values. Also at any given time alternating processors are idle. It is possible to use $w / 2$ processors in the network for a general band matrix with band width w. All the components of $y$ can be computed in ( $2 n+w$ ) time units


FIGURE 3.14: The linearly connected network for the matrix-vector multiplication problem shown in Figure 3.13

Step
Number
Comments


I


2



$Y_{1}$ is fed into the fourth processor initialized at 0 .
$x_{1}$ is fed into the first processor while $y_{1}$ is moved left one place (from now on the $x_{1}$ and $y_{1}$ keep moving right and left respectively).
$a_{11}$ enters the second processor where $Y_{1}-Y_{1}+a_{11} x_{1}$. Thus
$Y_{1}=a_{11} x_{1}$.
$a_{12}$ and $a_{21}$ enter the first and third processors, respectively, $y_{1}=a_{11} x_{1}+a_{12} x_{2}$ and $y_{2}=a_{21} X_{1}$.
$y_{1}$ is output $y_{2}=a_{21} x_{1}+a_{22} x_{2}$ $Y_{3}=a_{31} x_{1}$.
$y_{2}=a_{21} x_{1}+a_{22} x_{2}+a_{23} x_{3}$
$y_{3}=a_{31} x_{1}+a_{32} x_{2}$.
$\mathrm{y}_{2}$ is output
$y_{3}=a_{31} x_{1}+a_{32} x_{2}+a_{33} x_{3}$
$y_{4}=a_{42} x_{2}$

FIGURE 3.15: The first seven steps of the matrix-vector multiplication algorithm
using the above network. While the same $n$ components of $y$ needs $O(w n)$ units of time by using a sequential algorithm on a uniprocessor.

As another example, a systolic array may be constructed for the multiplication of two banded matrices. The matrix multiplication can be computed by the following recurrences:

$$
\begin{aligned}
& c_{i j}^{(1)}=0 \\
& c_{i j}^{(k+1)}=c_{i j}^{(k)}+a_{i k} b_{k j}, k=1,2, \ldots, n \\
& c_{i j}=c_{i j}^{(n+1)}
\end{aligned}
$$

Given two band matrices of bandwidths $w_{1}$ and $w_{2}$ respectively, then ( $w_{i} w_{2}$ ) processing cells are required to form a systolic array for matrix multiplication. The resultant product matrix has a bandwidth of $\mathrm{w}_{1}+\mathrm{w}_{2}-1$. Figure 3.16 shows an example of a band matrix multiplication application. Both matrices $A$ and $B$ have a bandwidth of size 4 and the resultant matrix $C=A . B$ has a bandwidth of 7 along its principal diagonal. In this example it is required that $w_{1} \times w_{2}=4 \times 4=16$ multiply cells are needed to construct the systolic array shown in Figure 3.17. The size of the array is determined by the bandwidths $w_{1}$ and $w_{2}$, independent of the dimension of the matrices.

The element of $A=\left(a_{i j}\right)$ and $B=\left(b_{i j}\right)$ matrices enter the array along the two diagonal data streams. The initial values of $C=\left(c_{i j}\right)$ entries are zeros. The outputs at the top of the vertical data stream give the product matrix $\left(c_{i j}^{o u t}=c_{i j}^{i n}+a_{i k} * b_{k j}\right.$ in each processing cell). Three data streams flow through the array in a pipelined fashion. If the time delay of each processing cell be one unit time. This systolic array can finish the band matrix multiplication in $T$ time units, where

$$
T=3 n+\min \left(w_{1}, w_{2}\right)
$$

Therefore, the computation time is linearly proportional to the dimension $n$ of the matrix.

When the matrix bandwidths increase to $\mathrm{w}_{1}=\mathrm{w}_{2}=\mathrm{n}$ (for dense matrices $A$ and $B)$, the time becomes $O(4 n)$, neglecting the input/output time delays (Hwang and Briggs [1984]). If one used a single divide-multiply processor to perform the same matrix multiplication $O\left(n^{3}\right)$ computation time would be needed. The systolic multiplier thus has a speed gain of $O\left(n^{2}\right)$. For large $n$, this improvement in speed is rather impressive.

A systolic array often represents a direct mapping of the computations onto processor arrays. It will be used as an attached processor of a host computer. The systolic array features the important properties of modularity, regularity, local interconnection, as well as a high degree of pipelining and highly synchronized multiprocessing. One problem however, is that the data movements in a systolic array are controlled by global timing-reference. In order to synchronize the activities in a systolic array, extra delays are often used to ensure correct timing.



To overcome the problems mentioned in systolic arrays, we need to take advantage of the control-flow locality, in addition to the dataflow locality. This permits a data-driven, self-timed approach to array processing. Conceptually, this approach substitutes the requirement of correct "timing" by correct "sequencing". This concept is used extensively in data flow computers and wavefront arrays. The wavefront array processor is conceived as a programmable, data-driven concurrent array processor aiming at solving a majority of matrix algorithms. Kung [1985] showed that, the derivation of a wavefront process consists of three steps: (i) the algorithms are expressed in terms of a sequence of recursions, (ii) each of the recursions is mapped to a corresponding computational wavefront, and (iii) the wavefronts are successively pipelined through the processor array.

Figure 3.18 shows a square, orthogonal $N \times N$ matrix array of processor elements with regular and local interconnections. Most matrix algorithms can be mapped naturally onto matrix arrays as shown in Figure 3.18.

The computational wavefront concept is used to create a smooth data movement in a localized communication network. A wavefront in the processing array will correspond to a mathematical recursion in the algorithm. Successive pipelining of the wavefronts through the computational array will accomplish the computation of all recursions. As an example consider the matrix multiplication which was discussed previously.

Let $A=\left(a_{i j}\right), B=\left(b_{i j}\right)$ and $C=A \times B$, all be $N \times N$ matrices. The two matrices $A$ and $B$ are decomposed such that matrix $B$ into rows $B_{j}$ and matrix $A$ into columns $A_{i}$, and hence,


FIGURE 3.18: The configuration of wavefront array processor ( $4 \times 4$ processing elements of the array)

$$
C=A_{1} B_{1}+A_{2} B_{2}+\ldots+A_{N} B_{N}
$$

The matrix multiplication can then be carried out in N recursions, executing,

$$
C^{(k)}=C^{(k-1)}+A_{k} B_{k} \quad, \text { with } C^{(0)}=0, \text { recursively }
$$

for $k=1,2, \ldots, N$.
To make it more clear, the computational wavefront for the first recursion in matrix multiplication is now shown in more detail. Suppose that initially, all the registers for the processing elements (PE's) are set to zero:

$$
c_{i j}^{(0)}=0 \text { for all } i, j
$$

The elements of matrix $A$ are stored in the memory modules to the left (in columns), and those of matrix $B$ in memory modules on the top (in rows). The process starts with $\mathrm{PE}(1,1)$ and

$$
c_{11}^{(1)}=c_{11}^{(0)}+a_{11} * b_{11}
$$

is computed. The computational activity then transmits to the neighbouring PE's $(1,2)$ and $(2,1)$ which will execute:

$$
c_{12}^{(1)}=c_{12}^{(0)}+a_{11} * b_{12}
$$

and

$$
c_{21}^{(1)}=c_{21}^{(0)}+a_{21} * b_{11}
$$

The next activity will be at PE's $(31),,(2,2)$ and $(1,3)$, thus creating a computation wavefront travelling down the processor array. Once the wavefront sweeps through all the cells, the first recursion is complete. An identical second recursion will be executed in parallel by pipelining a second wavefront immediately after the first one. For example, the ( 1,1 ) processor will execute

$$
\begin{gathered}
c_{11}^{(2)}=c_{11}^{(1)}+a_{12}+b_{21} \\
\vdots \\
c_{i j}^{(k)}=a_{i 1} * b_{l j}+a_{i 2} * b_{2 j}+\ldots+a_{i k} * b_{k j}
\end{gathered}
$$

The major characteristics of wavefront arrays are:
(i) Self-timed, data-driven computation, meaning that no global clock is needed.
(ii) Regularity, modularity and spatial locality of interconnections.
(iii) Effective pipelinability.

To conclude this section, one can often reduce the number of processors required by an algorithm if the matrix is known to be sparse or symmetric. For "sparse band matrices", which are band matrices whose non-zero entries appear only in a few of those lines in the band which are parallel
to the diagonal, then 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.

### 3.5 PARALLEL NUMERICAL ALGORITHMS

Parallelism may exist naturally in a numerical algorithm that is designed for a sequential computer and a parallel numerical algorithm can be created by recognizing the inherent parallelism in a sequential algorithm. Natural parallelism is present mainly in the algorithms of linear algebra and in the algorithms for the numerical computation of partial differential equations.

It is sometimes necessary to exploit hidden parallelism in a serially implemented algorithm. This is possible after a modification of the original algorithm, which usually requires either the introduction of redundant operators or the addition of some new ideas to the original method which were not contained in its serial implementation. Such algorithms are chaotic relaxation for the computation of linear equations, the computation of the eigenvalues of symmetric matrices by Jacobi's method, eliminating simultaneously several matrix elements and splitting the original set of ordinary differential equations into many subsets.

Finally, there also exists parallel algorithms specially designed for parallel computers. Their implementation on serial computers is of no value. They are algorithms for solving problems for which only complete serial algorithms have been available so far, such as recurrent relationships, elimination methods for the calculation of tridiagonal systems of linear equations, one-step iterations for calculating roots of non-linear equations, one-step methods for the computation of the initial values for ordinary difference equations, and the Gauss-Seidel iteration method. When computing these problems on a parallel computer, it was necessary to either implement the whole algorithm or design a new algorithm for their calculation.

Several surveys of parallel numerical algorithms have been introduced by different researchers. Miranker [1971] summarized some early work in numerical analysis, Sameh [1977] explained in general the numerical parallel algorithms, Heller [1978] gave a survey of parallel algorithms in numerical linear algebra, Feilemeier [1981] gave a detailed study for a wide area of numerical algorithms and finally, Schendel [1984] provided an introduction to the principles of parallel numerical analysis and certain recognized principles which are required for the development of parallel numerical algorithms.

In this section some of the parallel numerical algorithms will be considered.

### 3.5.1 Inherent Parallelism

As we mentioned earlier, one way to construct a parallel algorithm is to start with a serial algorithm and convert it into a parallel one. This principle which can be applied to most SIMD machines is to start with a serial algorithm and to convert it into a precedure which operates on vectors which can be carried out in parallel. As an example, consider the solution of an $n \times n$ system of linear equations with a lower triangular* coefficient matrix,

$$
\mathrm{Ax}=\underline{\mathrm{b}},
$$

where $\quad \underline{b}=\left(b_{1}, b_{2}, \ldots, b_{n}\right)^{T}, \quad \underline{x}=\left(x_{1}, x_{2}, \ldots, x_{n}\right)^{T}$,

$$
A=\left[\begin{array}{llcc}
a_{11} & & & \\
a_{21} & a_{22} & & \\
a_{31} & a_{32} & a_{33} & \\
\vdots & \vdots & \vdots & \\
a_{n 1} & a_{n 2} & a_{n 3} & \cdots \\
a_{n n}
\end{array}\right] \text {, and } A \text { is non-singular*. }
$$

[^3]\[

\left[$$
\begin{array}{llll}
a_{11} & & &  \tag{3.5.1.1}\\
a_{21} & a_{22} & & \\
1 & 1 & , & \\
1 & 1 & & \\
& \\
a_{n 1} & a_{n 2} & & a_{n n}
\end{array}
$$\right] \cdot\left[$$
\begin{array}{c}
x_{1} \\
x_{2} \\
1 \\
1 \\
1 \\
x_{n}
\end{array}
$$\right]=\left[$$
\begin{array}{c}
b_{1} \\
b_{2} \\
1 \\
1 \\
1 \\
b_{n}
\end{array}
$$\right]
\]

Schendel [1984] shows that, by using a serial computer $n^{2}$ arithmetic operations are necessary to effect the inversion. If for $i=1,2, \ldots, n$ we define

then it holds that $L_{n} L_{n-1} \cdots L_{1} A=I_{n}$, and thus

$$
\underline{x}=L_{n} L_{n-1} \cdots L_{1} b .
$$

Then we obtain the solution from the relation

$$
Y^{(1)}=\underline{b}, \quad Y^{(i+1)}=L_{i} Y^{(i)}, i=1,2, \ldots, n,
$$

where $Y^{(n+1)}=\underline{x}$. This relation represents the Gaussian elimination for (3.5.1.1). It can be shown that for $i=1,2, \ldots, n$

$$
L_{i} Y^{(i)}=\left(y_{1}^{(i)}, \ldots, y_{i}^{(i)} / a_{i i}, y_{i+1}^{(i)}-\frac{a_{i+1, i}}{a_{i i}} y_{i}^{(i)} \ldots, y_{n}^{(i)}-\frac{a_{n i}}{a_{i i}} y_{i}^{(i)}\right)^{T}
$$

If $n$ processors are available the solution $x$ can be calculated in approximately $3 n$ steps, that is $O(n)$.
The speed-up is, $S_{n}=\frac{n^{2} \text { (for serial) }}{3 n \text { (for parallel) }}=\frac{n}{3}$ and,
the efficiency, $E_{n}=\frac{S_{n}}{n}=\frac{n}{3 . n}=\frac{1}{3}$.
If, however, the number of processors available is $k$, where $k<n$, then, corresponding to $n$ serial steps, $\left\lceil\frac{n}{k}\right\rceil$ parallel steps are necessary for the calculation of ( $L_{i} y^{(i)}$ ), therefore, $O\left(\frac{n^{2}}{k}\right)$ steps are necessary, giving a speed-up of $\mathrm{S}_{\mathrm{k}}=\mathrm{O}(\mathrm{k})$ and an efficiency of $\mathrm{E}_{\mathrm{k}}=\mathrm{O}(1)$.

To make it more clear, in the serial procedure it is equivalent to setting, $A \underline{A}=\underline{b}$ to $A^{-1} A \underline{x}=A^{-1} b$ by the division of the rows by the diagonal terms and the reduction to zero of subdiagonal* terms in columns by multiplication and subtraction. Thus, to deal with the first row requires one division (by $a_{11}$ ). To reduce the $(2,1)$ element to zero involves replacing the second element of the $\underline{b}$ vector $b_{y} b_{2}-a_{21} *\left(\frac{b_{1}}{a_{11}}\right)$, that is a multiplication and subtraction. The processing of the first column thus involves $1+2(n-1)$ operations. For the second column and subdiagonal terms $1+2(n-2)$ operations are necessary. While the whole process requires $n$ divisions plus $2(1+2+3+\ldots+n-1)$ multiplications and subtractions, i.e., a total of $n+2 . \frac{1}{2}(n-1) n=n^{2}$ arithmetical operations in sequence.

$$
\begin{aligned}
& \text { For } n=4 \text {, the proposed procedure is as follows, } \\
& L_{1}=\left[\begin{array}{cccc}
-\frac{1}{a_{11}} & 0 & 0 & 0 \\
-\frac{a_{21}}{a_{11}} & 1 & 0 & 0 \\
-\frac{a_{31}}{a_{11}} & 0 & 1 & 0 \\
-\frac{a_{41}}{a_{11}} & 0 & 0 & 1
\end{array}\right],\left[\begin{array}{l}
b_{1} \\
b_{2} \\
b_{3} \\
b_{4}
\end{array}\right] \text { and } y_{2}=L_{1} y_{1}=\left[\begin{array}{l}
-\frac{b_{1}}{a_{11}} \\
-\frac{a_{21} b_{1}}{a_{11}}+b_{2} \\
-\frac{a_{31} b_{1}}{a_{11}}+b_{3} \\
-\frac{a_{41} b_{1}}{a_{11}}+b_{4}
\end{array}\right]
\end{aligned}
$$

[^4]By using 4 processors, the parallel operations are shown below. In the diagrams the symbols at the roots of the trees are the result of the operations illustrated.

Processor 1


The time required to execute all operations in parallel corresponds to a division (by $a_{11}$ ), multiplication and division, 3 time units. Note that $x_{1}=y_{21}$. The next step uses the matrix

$$
L_{2}=\left[\begin{array}{cccc}
1 & 0 & 0 & 0 \\
0 & \frac{1}{a_{22}} & 0 & 0 \\
0 & -\frac{a_{32}}{a_{22}} & 1 & 0 \\
0 & -\frac{a_{42}}{a_{22}} & 0 & 1
\end{array}\right]
$$

to operate on $y_{2}=\left(y_{21}, y_{22}, y_{23}, y_{24}\right)$ to get $y_{3}$. Since $x_{1}=y_{21}$ only three components, $Y_{32}, Y_{33}$ and $y_{34}$ are needed in the next step and only three processors need to be used.

Processor 2

$\frac{1}{a_{22}} * y_{22}-\frac{a_{32}}{a_{22}} * y_{22}+y_{23}$


Again, 3 time units have been used and $x_{2}=y_{32}$. The remaining two steps can be shown schematically as, Processor


This gives $\mathrm{x}_{3}=\mathrm{y}_{43^{\circ}}$ Finally,

and $x_{4}=y_{54}$. All steps have used 3 units of time except the last which requires only 2. Thus, for general $n$ the number of time steps are $3 n-1$.

### 3.5.2 Vector Iteration Method

In a vector iteration method, a direct (non-iterative) serial algorithm is substituted by an iterative parallel algorithm. The speed-up factor of the parallel version depends on the ratio of the steps needed in the direct version to those required by the iteration. Heller [1978] gives an example for the triangular decomposition of a tridiagonal matrix A.

where

The $u_{i}$ elements are calculated from the scheme

$$
u_{1}=d_{1}, \quad u_{i}=d_{i}-e_{i} f_{i-1} / u_{i-1}, 2 \leqslant i \leqslant n
$$

The $m_{i}$ can then be calculated in parallel

$$
m_{i}=e_{i} / u_{i-1} \quad, i=2,3, \ldots, n
$$

The iteration method can be used to convert the serial equation (3.5.2.1) to a parallel version as:

$$
\begin{aligned}
& u_{i}^{(0)}=d_{i} \\
& u_{i}^{(j)}=d_{i}-e_{i} f_{i-1} / u_{i-1}^{(j-1)} \quad, i=1,2, \ldots, n,
\end{aligned}
$$

where $u^{(j)}$ is the jth iterate.
This parallelism can be a reasonable one if the computer can carry out operations with vectors of $n$ components faster than $n$ scalar operations. Moreover the number of iterations required must naturally be significantly less than $n$.

### 3.5.3 Recurrence Relations

In numerical analysis, the solution of a problem is often expressed as a sequence of evaluations $x_{1}, x_{2}, \ldots, x_{n}, \ldots$, where each $x_{i}(i=1,2, \ldots, n)$ may depend on one or more $x_{j}$ s where $j<i$. The equations describing this dependence are called "recurrence relations". Because the definition of the recurrence algorithm is given in terms of a sequential evaluation which presents a special problem for a parallel computer, the problem can be re-phrased so as to allow the parallel evaluation to be carried out at the expense of introducing extra arithmetic operations. As an example, we consider the evaluation of the partial sums of a sequence of numbers.

The general linear first-order recurrence can be expressed as the evaluation of the sequence $x_{j}$ from the recurrence relation,

$$
\begin{equation*}
x_{j}=a_{j} x_{j-1}+d_{j}, \quad j=1,2, \ldots, n \tag{3.5.3.1}
\end{equation*}
$$

The partial sum $S_{i}, i=1,2, \ldots, n$ can be evaluated as a special case of (3.4.3.1), and defined as,

$$
\begin{equation*}
S_{i}=\sum_{j=1}^{i} a_{j}, \quad i=1,2, \ldots, n, \tag{3.5.3.2}
\end{equation*}
$$

where $s_{j}$ is the sum of the first $j$ numbers in the sequence $d_{1}, d_{2}, \ldots, d_{n}$. The partial sums may be evaluated simply from the recurrence

$$
s_{i}=s_{i-1}+a_{i}, \quad i=1,2, \ldots, n
$$

The sequential sum method of evaluation may be realised with ( $n-1$ ) additions and ( $n-1$ ) routing operations as shown in Figure 3.19, for the case $\mathrm{n}=8$.

Hockney and Jesshope [1981] describe an alternative parallel approach to the partial sum (3.5.3.2) which is known as the "cascade partial sum" and is shown in Figure 3.20 for the case $n=8$.


FIGURE 3.19: The sequential evaluation of partial sums of eight numbers. The vertical axis is time, the horizontal axis is storage location or processing element number.

An array of $n$ variables is first loaded with the data (a's). In level one, a copy of the variables is shifted one place to the right and added to the unshifted variables in order to form the sum of the adjacent variables. At the next level, the process is repeated but with a shift of two places to the right, thereby producing sums of groups of four numbers. As the shifts are made, zeros are brought in from the left as required. In general, at the $\ell$ th time level a shift of $2^{\ell}$ places is made and at level $\ell=\log _{2} n$ the variables contain the required partial sums.

The cascade partial sum method requires $\log _{2} n$ additions with parallelism $n$, and ( $n-1$ ) routing operatings with parallelism $n$. The sequential evaluation of the general first-order recurrence (3.5.3.1) requires, 2 n arithmetic operations with parallelism 1 and n routings with parallelism 1. As shown in Figure 3.21, variables linked by a brace are stored in the same PE. One PE is used to evaluate each term of the recurrence.

The equivalent parallel algorithm to the cascade sum method is known as "cyclic reduction", and has a wide application in numerical analysis, particularly when one is trying to introduce parallelism into a problem. Figure 3.22 shows the evaluation tree of the cyclic reduction method to linear first-order sequences, for $n=8$.

There exists many parallel numerical algorithms specially designed for parallel computers, among those algorithms are the new parallel algorithms for partial differential equations, Evans [1984]; the solution of systems of linear equations. Chen and Kuck [1975]. Heller [1978] and Borodin and Munro [1975] have also shown that if $p=O\left(n^{3}\right)$ processors are available, a triangular system of $n$ equations
$\ell=3$

$\ell=2$


FIGURE 3.20: The cascade sum method of forming partial sums. If only the total sum $S_{8}$ is required, then only the operations shown with dotted lines are carried out.


FIGURE 3.21: The sequential evaluation of the general first-order recurrence


FIGURE 3.22: The cyclic reduction of the general first-order recurrence relation
of the form $L \underline{x}=\underline{b}$ can be resolved in $O\left(\log _{2}^{2} n\right)$ time steps. Sameh and Brent [1977] represent algorithms for the solution of a dense triangular system of equations. Also, parallel versions of well known sequential numerical algorithms have been implemented by different researchers, such as, the parallel LR-algorithm, the parallel Gauss algorithm and the parallelisation of iterative algorithms such as the Jacobi, GaussSeidel and SOR algorithms.

### 3.6 PARALLEL NON-NUMERICAL ALGORITHMS

The algorithms derived from non-numerical applications are an area which is widely applied and from which the benefits of parallelism are very large. An example of a parallel non-numerical algorithm that has been investigated by different researchers are the sorting of a given set of numbers, the search for a given argument, the merging of strings of numbers, deleting an element from a linked list and counting the number of elements in a linked list, etc.

Sorting is an important application on a sequential computer and its implementation on SIMD and MIMD parallel computers have been studied by many researchers. In the first example, the implementation of the parallel non-numeric algorithm on MIMD type machines will be considered. Yousif [1983] presents many non-numerical algorithms for MIMD computers, among these are different merging and sorting algorithms. The basic principles of the 2-way merge algorithm is given by Knuth [1973]. The parallel implementation of this algorithm is performed on the $M$ sorted subsets of size $\left(\frac{N}{M}\right)$ each, where $N$ is the size of input, $N$ is divisible by $M$ and $M$ is a power of 2 . This merge algorithm can be completed in (logM) steps where parallelism is introduced within each step and not amongst the steps as is shown in Figure 3.23.

Steps/subset

1

2

3


FIGURE 3.23: The parallel 2-way merge algorithm

In Figure 3.23, each step can be performed in parallel, where each two neighbouring subsets are merged by one process to form a subset of size $\left(\frac{2 N}{M}\right)$. Also from Figure 3.23 it can be noticed 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.

Yousif [1983] shows that the total complexity of the 2-way merge algorithm in one processor is,

$$
t_{1 M}=N \log M-M+1
$$

and the total complexity of the 2 -way merge algorithm in P processors is,

$$
t_{P M} \leqslant \frac{N}{P} \log \frac{M}{P}+\frac{2 N}{P}(P-1)-\frac{M}{P}+\log \left(\frac{M}{P}\right)+1
$$

The speed-up ratio for the merge algorithm is,

$$
S_{\text {merge }}(M)=\frac{t_{1 M}}{t_{P M}} \text {, which is of } O\left(\frac{P \log M}{\log \left(\frac{M}{P}\right)+2 P-2}\right)
$$

Yousif [1983] implements another parallel non-numerical algorithm using the above parallel merge algorithm in its final stage. This is the well known sequential sorting algorithm, the bubble sort (Knuth [1973]). The serial "bubble sort" proceeds by comparing and exchanging pairs of adjacent items. In order to sort an array ( $x_{1}, x_{2}, \ldots, x_{n}$ ), $(n-1)$ comparison-exchanges $\left(x_{1}, x_{2}\right)^{\prime},\left(x_{2}, x_{3}\right), \ldots,\left(x_{n-1}, x_{n}\right)$ are performed. This results in placing the maximum at the right end of the array. After this first step, $x_{n}$ is discarded, and the same "bubble" sequence of comparison-exchanges is applied to the reduced array ( $x_{1}, x_{2}, \ldots, x_{n-1}$ ). By iterating ( $n-1$ ) times, the entire sequence is sorted. Knuth [1973] shows that to run the algorithm on a sequential machine (i.e. one
processor) where $N$ is the number of elements to be sorted, needs an average,
and
$C \cong \frac{1}{2}\left(N^{2}-N \ell n N-N\right)$ comparisons
$E \cong \frac{1}{4}\left(N^{2}-N\right)$ exchanges
i.e., the total running-time of the algorithm, $t_{1}=\frac{3}{4} \mathrm{~N}^{2}-\frac{1}{2} \mathrm{~N} \ln \mathrm{~N}-\frac{3}{4} \mathrm{~N}$ that is the sequential bubble sort algorithm is of $O\left(N^{2}\right)$.

Yousif [1983] present a parallel bubble sort version on an MIMD type machine. In that implementation, the input set to be sorted is partitioned into as many subsets as the system allows. Then for each subset the sequential bubble sort method is applied. The final step will be the merging step, where all the sorted subsets are merged to determine the linear ordering of their unions by means of pairwise comparisons between the subset elements. It was shown that, if the input set $N$, is divided into $M$ subsets ( $M \geqslant P$, where $P$ is the number of available processors) each contains $N / M$ elements, the total time to run the algorithm on one processor is:

$$
\begin{aligned}
t_{1 S} & =M\left[\frac{3}{4}\left(\frac{N}{M}\right)^{2}-\frac{1}{2} \frac{N}{M} \ln \left(\frac{N}{M}\right)-\frac{3}{4} \frac{N}{M}\right] \\
& =\frac{3}{4} \frac{N^{2}}{M}-\frac{1}{2} N \ln \left(\frac{N}{M}\right)-\frac{3}{4} N .
\end{aligned}
$$

Meanwhile, when this algorithm is run in parallel with $P$ processors, $\left\lceil\frac{M}{\mathrm{P}}\right\rceil$ subsets have to be carried out by each processor. Thus, the total time needed is,

$$
\begin{aligned}
t_{P S} & =\left\lceil\frac{M}{P}\right\rceil\left[\frac{3}{4} \frac{N^{2}}{M^{2}}-\frac{1}{2} \frac{N}{M} \ln \left(\frac{N}{M}\right)-\frac{3}{4} \frac{N}{M}\right] \\
& \leqslant \frac{3}{4} \frac{N^{2}}{M P}-\frac{1}{2} \frac{N}{P} \ln \left(\frac{N}{M}\right)-\frac{3}{4} \frac{N}{P}+1
\end{aligned}
$$

The total time $\left(T_{1}\right)$ to run the bubble algorithm on one processor is equal to the corresponding times for the sorting and merging sections.

Then,

$$
\begin{aligned}
T_{1} & =t_{1 S}+t_{1 M} \\
& =\frac{3}{4} \frac{N^{2}}{M}-\frac{1}{2} N \ln \left(\frac{N}{M}\right)-\frac{3}{4} N+N \log M-M+1 .
\end{aligned}
$$

For the parallel implementation, when the bubble sort algorithm is run on $P$ processors, the total parallel time ( $T_{p}$ ) will be equal to the corresponding times for sorting and merging. Then,

$$
\begin{aligned}
T_{p} & =t_{P S}+t_{P M} \\
& \leqslant \frac{3}{4} \frac{N^{2}}{M P}-\frac{1}{2} \frac{N}{P} \ln \left(\frac{N}{M}\right)-\frac{3}{4} \frac{N}{P}+\frac{N}{P} \log \left(\frac{M}{P}\right)+\frac{2 N}{P}(P-1)-\frac{M}{P}+\log \left(\frac{M}{P}\right)+2 .
\end{aligned}
$$

The speed-up ratio, $S_{\text {total }}(M)=\frac{T_{1}}{T_{p}}$ which is of $O\left(\frac{P \log M}{\log \left(\frac{M}{P}\right)+2 P-2}\right.$.
For SIMD type parallel computers, Baudet and Stevenson [1978] present a sort algorithm which is based upon a generalization of the odd-even transposition sort (Knuth [1973]). The serial odd-even transposition sort can be considered as an algorithm for sorting $K$ elements using $K$ processors in $K$ steps of parallel "comparison exchanges". The algorithm works as follows:

Let $a_{1}, a_{2}, \ldots, a_{k}$ be the sequence to be sorted. In the first step, for $i=1,3, \ldots, 2\lfloor k / 2\rfloor-1$, processor $P_{i}$ compares elements $a_{i}$ and $a_{i+1}$ and if $a_{i}>a_{i+1}$ the two elements are exchanged. In the second step, the same comparison exchanges are executed for $i=2,4, \ldots, 2\lfloor(k-1) / 2\rfloor$. Steps $3,5, \ldots$ are the repetitions of step 1 , and steps $4,6, \ldots$, are repetitions of step 2. A generalization of the above algorithm to partially sorted rsequences is shown below. In the first step for $i=1,3, \ldots, 2\lfloor k / 2\rfloor-1$, processor $P_{i}$ merges the two subsequences $S_{i}$ and $S_{i+1}$ and then assigns to $S_{i}$ the first half of the resulting merged sequence and assigns to $S_{i+1}$ the second half. For the second step, the same operations are
executed but for $i=2,4, \ldots, 2\lfloor(k-1) / 2\rfloor$. Again steps $3,5, \ldots$, are the repetitions of step 1 and steps $4,6, \ldots$, are repetitions of step 2 , as shown below in Figure 3.24 .

|  |  | Step 1 |  | Step 2 |  | Step 3 |  | Step 4 |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| S1: | 43,54,63 | P1 | 28,43,54 |  | 28,43,54 | P1 | 17,25,28 |  | 17,25,28 |
| S2 : | 28,72,79 |  | 63,72,79 | ${ }^{\text {P2 }}$ | 17,25,32 |  | 32,43,54 | $\xrightarrow{\text { P2 }}$ | 32,43,47 |
| S3: | 32,47,84 | P3 | 17,25,32 |  | 63,72,79 | P3 | 47,63,66) |  | 54,63,66 |
| S4: | 17,25,66 |  | 47,66,84 |  | 47,66,84) |  | 72,79,84 |  | 72,79,84 |

FIGURE 3.24: Four steps of parallel "merging-splittings" for the partially sorted 3-sequence numbers

The total execution time of the algorithm is:

$$
T=(n \log n) / k+O(n)
$$

For a sequential computer, the minimum number of comparisons required for sorting a sequence of $n$ numbers is asymptotically nlogn. Therefore, when $k$ is smaller than logn, the asymptotic speed-up ratio of the algorithm over the optimal sequential algorithm is $k$, which is optimal. In particular, when $k=\operatorname{logn}$, the ratio of this parallel algorithm to the optimal sequential algorithm is of order logn, the number of processors. On the other hand, when $k$ is greater than logn, the total execution time required for the algorithm is asymptotically linear in $n$.

In Thompson and Kung [1977] two SIMD algorithms are presented to sort $n^{2}$ elements on an $n \times n$ mesh-connected processor array. In their model, they assume a parallel computer with $N=n \times n$ identical processors, the interconnections between the processors are defined by the following two dimensional array:

where the P's denote the processors. That is, each processor is connected to all its neighbours. Processors at the boundary have two or three rather than four neighbours. The two timing factors used in the evaluation of the algorithm complexity are the routing time ( $t_{R}$ ) and the comparison time ( $t_{C}$ ). Routing time ( $t_{R}$ ), is the time required to move one item from a processor to one of its neighbours. While comparison time $\left(t_{C}\right)$ is the time required for one comparison step. Concurrent data movement is allowed as long as it is all in the same direction, and also up to $N$ comparisons may be performed simultaneously. This means that a comparison-interchange step between two items in adjacent processors can be done in time $\left(2 t_{R}+t_{C}\right)$ time units (route left, compare, route right). A mixture of horizontal and vertical comparison interchanges requires at least $\left(4 t_{R}+t_{C}\right)$ time units. The $N$ multiprocessors are indexed according to a pre-specified rule. The indexing rules considered are the row-major, the snake-like row-major and the shuffled row-major rules as shown in Figure 3.25. If it is assumed that $N$ keys with arbitrary values are initially loaded in the N processors, each receiving exactly one key, the sorting problem consists of moving the ith smallest key to the processor indexed by $i$,
for $i=1,2, \ldots, N$. The choice of a particular indexing scheme depends upon how the sorted elements will be used.

(a) Initial loading pattern before sorting

(c) Sorted pattern with shuffled row-major indexing
(d) Sorted pattern with snake-like row-major indexing

FIGURE 3.25: Sorting patterns with respect to three ways of indexing the $P^{\prime} s$.

In Thompson and Kung [1977], two algorithms are presented that perform this array sort in $O(n)$ comparisons and moves. In the first algorithm, the odd-even merge sort [Batcher [1968], Knuth [1973]) on a linear array has been generalized to a square array of processing elements. The second algorithm uses a bitonic sort (Batcher [1968], Knuth [1973]) and orders the keys with shuffled row-major indexing. Nassimi and Sahni present an $O(n)$ algorithm to sort $n^{2}$ elements
on an $n \times n$ mesh-connected processor array. This algorithm sorts the $n^{2}$ elements into row-major order and is an adaptation of Batcher's bitonic sort.

Implementations of more parallel non-numerical sorting and searching algorithms will be discussed in Chapter 7.

### 3.7 PERFORMANCE OF PARALLEL COMPUTER ALGORITHMS

In the previous sections different methods for parallel algorithm design have been considered. However in a study of parallel computer algorithms, we need some measure of efficiency to evaluate the algorithms. An appropriate measure for specific problems is the speed-up ratio and the efficiency of the algorithm. If $T_{p}$ denotes the computation time on a computer with $P$ processors, and $T_{1}$ to denote the computation time of a sequential computer (uniprocessor). Then the speed-up ( $S_{p}$ ) of a $P$ processor computer over a sequential computer is defined as:

$$
\begin{equation*}
S_{p}=\frac{T_{1}}{T_{p}} \geqslant 1 \tag{3.7.1}
\end{equation*}
$$

and the efficiency ( $E_{p}$ ) is defined as:

$$
\begin{equation*}
E_{p}=\frac{S_{p}}{p} \leqslant 1 \tag{3.7.2}
\end{equation*}
$$

Where $S_{p}$ is the maximum speed-up using $P$ simultaneous processors and $E_{p}$ measures the utilisation of the parallel machine. The longer that processors are idle, or carry out extra calculations introduced through the parallelisation of the problem, the smaller $E_{p}$ becomes. It can be verified that these definitions are consistent with the uniprocessor case when $\mathrm{p}=1$. To achieve a fair comparison, we always compare the best serial algorithm for the computation with the best parallel algorithm, even when the two algorithms are quite different.

Stone [1973] indicates that for a computer system with $N$ processors, the ideal speed-up ratio is $N$, but this is hardly ever achieved. Computations that are very well-suited to parallel computer systems have a speed-up of $K N$, where $K$ is a constant near unity, but strictly less than unity. The best speed-up ratio are linear in $N$.

To compare two parallel algorithms for the same problem Schendel [1984] introduces the following measure of effectiveness $F_{p}$ as:

$$
\begin{equation*}
F_{p}=S_{p} / C_{p} \tag{3.7.3}
\end{equation*}
$$

where,

$$
\begin{equation*}
C_{p}=P T_{p} \tag{3.7.4}
\end{equation*}
$$

measures the 'cost' of the algorithm. It can be noticed that

$$
\begin{equation*}
F_{p}=\left(S_{p} / p T_{p}\right)=E_{p} / T_{p}=E_{p} S_{p} / T_{1} \leqslant 1 \tag{3.7.5}
\end{equation*}
$$

$F_{p}$ is thus a measure of both speed-up and efficiency. An effective parallel algorithm is the one that maximises $F_{p}$.

As an example, consider the computation of the sum,

$$
A=\sum_{1}^{16} a_{i}
$$

It can be seen that 15 additions are required to compute $A$ with a single processor, and we have $T_{1}=15$ time units. If two processors were available we would form the two sums,

$$
b_{1}=a_{1}+a_{2}+\ldots+a_{8}, b_{2}=a_{9}+a_{10}+\ldots+a_{16}
$$

simultaneously, requiring 7 time units, and then form,

$$
c=b_{1}+b_{2}
$$

at the next stage, requiring a further time unit. Thus $A$ would be obtained in $T_{2}=7+1=8$ time units.

Given 3 processors A could be formed in the following three stages:

$$
\begin{aligned}
& b_{1}=a_{1}+a_{2}+a_{3}+a_{4}+a_{5}, b_{2}=a_{6}+a_{7}+a_{8}+a_{9}+a_{10} \\
& b_{3}=a_{11}+a_{12}+a_{13}+a_{14}+a_{15} \\
& c_{1}=b_{1}+b_{2}, c_{2}=b_{3}+a_{16} ; \\
& d_{1}=c_{1}+c_{2}=A .
\end{aligned}
$$

This requires $\quad T_{3}=4+1+1=6$ time units

For 4 and 8 processors the following procedure and their corresponding times are shown:

for | $\mathrm{p}=4$ |  |
| ---: | :--- |
| $\mathrm{~b}_{1}$ | $=\mathrm{a}_{1}+\ldots+\mathrm{a}_{4}, \mathrm{~b}_{2}=a_{5}+\ldots+\mathrm{a}_{8}, b_{3}=a_{9}+\ldots+\mathrm{a}_{12}, b_{4}=a_{13}+\ldots+\mathrm{a}_{16}$ |
| $c_{1}$ | $=b_{1}+b_{2}, \quad c_{2}=b_{3}+b_{4}$ |
| $d_{1}$ | $=c_{1}+c_{2}$, |

giving, $\quad T_{4}=5$ time units,
and for $p=8$,

$$
\begin{align*}
& b_{1}=a_{1}+a_{2}, b_{2}=a_{3}+a_{4}, \ldots, b_{8}=a_{15}+a_{16}  \tag{1}\\
& c_{1}=b_{1}+b_{2}, c_{2}=b_{3}+b_{4}, \ldots, c_{4}=b_{7}+b_{8}  \tag{1}\\
& a_{1}=c_{1}+c_{2}, d_{2}=c_{3}+c_{4}  \tag{1}\\
& A=d_{1}+d_{2} \tag{1}
\end{align*}
$$

giving, $\quad T_{8}=4$ time units.
A table of the performance measures can now be constructed:

| $p$ | $T_{p}$ | $C_{p}$ | $S_{p}$ | $E_{p}$ | $F_{p} T_{1}=S_{p} E_{p}$ |
| :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 15 | 15 | 1 | 1 | 1 |
| 2 | 8 | 16 | 1.88 | 0.94 | 1.76 |
| 3 | 4 | 18 | 2.5 | 0.83 | 2.08 |
| 4 | 5 | 20 | 3 | 0.75 | 2.25 |
| 8 | 4 | 32 | 3.75 | 0.47 | 1.76 |

The table shows that with increasing $P, S_{p}$ increases steadily while $E_{p}$ decreases. $F_{p} T_{1}$, however has a maximum when $P=4$ which indicates that $\mathrm{P}=4$ is the optimal choice of the number of processors for this calculation.

The parameters introduced above give one measure for the assessment of a parallel algorithm. Other aspects for consideration are stability and the analysis of errors.

## Chapter 4

## PARALLEL IMPLEMENTATIONS OF THE

## FOUR-POINT AND NINE-POINT EXPLICIT-BLOCK

ITERATIVE METHODS

### 4.1 INTRODUCTION

The solution to a variety of scientific problems can often be obtained by solving a set of linear equations. Different methods have been suggested to solve such a set of equations, and among these methods are the point and the block (group) iterative methods (Young, 1971). In this chapter the point and block (group) iterative methods are both represented and discussed, and their mathematical foundations are shown. Since the work done in this thesis is mainly applied to a MIMD parallel computer, so the asynchronous aspects of these methods are emphasised and considered. Other parallel iterative methods for solving a system of linear equations are also considered and are discussed in this chapter.

In this chapter, two methods for solving a linear system of equations are implemented in parallel with the use of the acceleration or overrelaxation parameter, $w$. The first method involving a block of nine points is called the nine-point block method, while the second method involves a block of four points, and is called the four-point block method. The basic derivation of both the sequential 4-point and 9-point block methods are shown with the computational amount of the work involved. On the other hand, different parallel implementations of the 9 -point and 4-point block methods by different orderings are also developed in this chapter. Two different parallel schemes are used in these implementations, these are the synchronous and asynchronous schemes.

The performance analysis of the 9 -point and 4 -point block methods with different implementations are presented also. It is well known that different algorithmic designs produce different timing results and

[^5]
### 4.2 BASIC DEFINITIONS

In this section we introduce some basic properties and definitions of matrices which will be used later in this chapter.

A system of $m$ linear equations with $n$ unknowns has the general form,

$$
\begin{align*}
& a_{11} x_{1}+a_{12} x_{2}+\ldots+a_{1 n} x_{n}=b_{1} \\
& a_{21} x_{1}+a_{22} x_{2}+\ldots+a_{2 n} x_{n}=b_{2} \\
& \vdots  \tag{4.2.1}\\
& \vdots \\
& a_{m 1} x_{1}+a_{m 2} x_{2}+\ldots+a_{m n} x_{n}=b_{m}
\end{align*}
$$

The right hand sides $b_{i}(i=1,2, \ldots, m)$ and the coefficients $a_{i, j}$, $(i=1,2, \ldots, m ; j=1,2, \ldots, n)$ are given numbers with the unknown vector $x_{i}, i=1,2, \ldots, n$. The problem is to $f$ ind, if possible, numbers $x_{j}$ $(j=1,2, \ldots, n)$ such that the $m$ equations (4.2.1) are satisfied simultaneously.

A matrix is a rectangular array of numbers arranged in rows and colums, and usually square brackets are used to denote the extent of the array. An individual number occurring in a matrix is called an element of the matrix. The coefficient of (4.2.1) form a matrix; called A (matrices will be denoted by a capital letter) which can be written as:

$$
A=\left[\begin{array}{cccc}
a_{11} & a_{12} & --- & a_{1 n}  \tag{4.2.2}\\
a_{21} & a_{22} & -\cdots- & a_{2 n} \\
1 & 1 & & 1 \\
1 & 1 & 1 \\
1 & 1 & & a_{m n}
\end{array}\right]
$$

The matrix $A$ in (4.2.2) has $m$ rows and $n$ columns and $A$ is said to be of order $m \times n$. If a matrix has both $n$ rows and $n$ columns it is said to be square of order $n$. If a matrix has only one row, it is known as a
row vector, and a matrix having only one column is called a column vector. Both row vector and column vectors are known as vectors for short and written by an underlined single lower case letter.

If $A$ is a matrix of order $m \times n$ then the element in the ith row and jth colum of matrix $A$ may be denoted by $a_{i j}$ and matrix $A$ may be denoted by

$$
A=\left(a_{i, j}\right)
$$

From (4.2.1) both $b_{i},(i=1,2, \ldots, m)$ and unknowns $x_{j}(j=1,2, \ldots, n)$ form the vectors,


We say that $\underline{b}$ is an $m$-vector, $\underline{x}$ is an $n$-vector and the system of equations (4.2.1) simply can be written as,

$$
\begin{equation*}
\mathrm{Ax}=\underline{\mathrm{b}} \tag{4.2.4}
\end{equation*}
$$

If $A=\left(a_{i, j}\right)$ is a square matrix of order $n$, then we call its elements $a_{i, i}, i=1,2, \ldots, n$ the diagonal entries of $A$, and all other elements are called off-diagonal. All elements $a_{i, j}$ of $A$ with $i<j$ are called superdiagonal, all entries $a_{i, j}$ with $i>j$ are called subdiagonal as shown in Figure 4.1.


FIGURE 4.1

A square matrix $A$ is said to be diagonal, if all its off-diagonal elements are zero (i.e., $a_{i, j}=0$ if $i \neq j$ ). 'If all subdiagonal entries of the square matrix A are zero, we call A an upper triangular matrix (i.e. $a_{i, j}=0$ if $i>j$ ), while if all superdiagonal entries of $A$ are zero, then $A$ is called Zower triangular (i.e. $a_{i, j}=0$, if $j>i$ ). We call A an unit upper triangular matrix if it is upper triangular with diagonal elements equal to unity, while A is called an unit lower triangular if it is lower triangular with diagonal elements equal to unity.

If most of the elements $a_{i, j}$ are non-zero, then the matrix $A$ is a dense matrix. On the other hand, if most of the elements $a_{i, j}$ of $a$ matrix $A$ are zero then $A$ is said to be a sparse matrix. A matrix whose elements are all zero is known as a null or zero matrix. If $A$ is a diagonal matrix of order $n$, has all its diagonal elements equal to 1 , then $A$ is called the identity matrix of order $n$ and denoted by $I$. The inverse of a given matrix $A$ denoted by $A^{-1}$ and such that,

$$
A^{-1} A=A A^{-1}=I
$$

where $I$ is the identity matrix.
If $A=\left(a_{i, j}\right)$ is a matrix of order $m \times n$ then the transpose of $A$, denoted by $A^{T}$, is the matrix of order $n \times m$ such that $A^{T}=\left(a_{j, i}\right)$. This means that the element appearing in the ith row and jth column of $A^{T}$ is the same as the element in the jth row and ith column of $A$. This implies that $A^{T}$ is obtained from $A$ merely by interchanging rows and columns. The matrix $A=\left(a_{i, j}\right)$ is said to be symmetric if it conicides with its transpose, that is, if

$$
A^{T}=A
$$

and is said to be anti-symmetric (or skew-symmetric) if

$$
A^{T}=-A
$$

The determinant of a square matrix $A$ will be denoted by $\operatorname{det}(A)$ or $|A|$. A square matrix is said to be $\operatorname{singular}$ if $\operatorname{det}(A)=0$; if $\operatorname{det}(A) \neq 0$ it is said to be non-singular. The matrix $A=\left(a_{i, j}\right)$ is diagonally dominant if

$$
\begin{equation*}
\left|a_{i, j}\right| \leqslant \sum_{\substack{i=1 \\ j \neq i}}^{n}\left|a_{i, j}\right|, \text { for all } 1 \leqslant i \leqslant n \tag{4.2.5}
\end{equation*}
$$

in other words, each entry of the main diagonal is greater or equal to the sum of the other entries of the row holding that particular entry. A is said to be strictly diagonally dominant if strict equality holds for all $1 \leqslant i \leqslant n$ in the equality (4.2.5).

The matrix $A=\left(a_{i, j}\right)$ is said to be band matrix if $a_{i, j}=0$ for $|i-j|>m$ (i.e. "bandwidth" $2 \mathrm{~m}+1$ since this is the number of non-zero diagonals in the band). As an example, if $a_{i, j}=0$ for $|i-k|>1, i . e .$, all elements are zero except for the main diagonal and sub- and super-diagonals, the matrix $A$ is said to be tridiagonal. In general, if there are $m$ nonzero diagonals immediately below and $\mathrm{m}_{2}$ 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" $p=m_{1}+m_{2}+1$.

## Eigenvalues and Eigenvectors

If $A$ is an ( $n \times n$ ) matrix and $x \neq 0$ is a vector of order $n$. If there exists a scalar $\lambda$ such that,

$$
\begin{equation*}
\underline{A x}=\lambda \underline{x}, \tag{4.2.6}
\end{equation*}
$$

then $x$ is said to be an eigenvector of $A$ with corresponding eigenvalue $\lambda$.

The system (4.2.6) can be written as,

$$
\begin{equation*}
(A-\lambda I) \underline{x}=\underline{0}, \tag{4.2.7}
\end{equation*}
$$

where $I$ is the identity matrix. The non-trivial solution $x \neq 0$ to the system (4.2.7) exists if and only if the matrix of the system is singular, i.e.,

$$
\begin{equation*}
\operatorname{det}(A-\lambda I)=0 . \tag{4.2.8}
\end{equation*}
$$

Equation (4.2.8) is known as the characteristic equation of A .
Expanding equation (4.2.8) leads to an explicit polynomial equation, the roots of which give all the eigenvalues $\lambda_{1}, \lambda_{2}, \ldots, \lambda_{n}$. Substituting each $\lambda_{r}$ into the equation (4.2.8) gives the $n$ sets of equations

$$
\begin{equation*}
\left(A-\lambda_{r} I\right) x^{(r)}=0, \tag{4.2.9}
\end{equation*}
$$

which when solved give the eigenvector $\mathrm{X}^{(r)}$.

Theorem 4.2.1 (Bell [1975])
If $A$ is a triangular matrix then its eigenvalues are its diagonal elements.

Proof:
First we take the case when A is lower triangular; an exactly similar proof holds for upper triangular A. Let,

To determine the eigenvalues from the characteristic equation:

$$
A-\lambda I=\left[\begin{array}{cccc}
a_{11}-\lambda & 0 & 0 \cdots-\cdots-0 \\
a_{21} & a_{22}-\lambda & 0-\cdots-\cdots & 0 \\
a_{31} & a_{32} & a_{33}-\lambda \cdots-\cdots-0 \\
\vdots & & & a_{n n}-\lambda
\end{array}\right]
$$

and thus

$$
\operatorname{det}(A-\lambda I)=\left(a_{11}-\lambda\right)\left(a_{22^{-\lambda}}^{-\lambda}\right)\left(a_{33}-\lambda\right) \ldots\left(a_{n n}-\lambda\right) .
$$

It follows from the characteristic equation is just

$$
\left(a_{11}-\lambda\right)\left(a_{22}-\lambda\right)\left(a_{33}-\lambda\right) \ldots\left(a_{n n}-\lambda\right)=0
$$

with roots $a_{11}, a_{22}, a_{33}, \ldots, a_{n n^{\prime}}$ which proves the required result. Theorem 4.2.2

If $A$ is a square matrix of order $n$ with eigenvalues $\lambda_{1}, \lambda_{2}, \ldots, \lambda_{n}$ then the sum and product of all the eigenvalues are given by,

$$
\begin{align*}
& T_{r}(A)=\sum_{i=1}^{n} \lambda_{i}  \tag{i}\\
& \operatorname{det}(A)=\prod_{i=1}^{n} \lambda_{i} . \tag{ii}
\end{align*}
$$

Proof
(see Bell [1975], page 153).

## Vector and Matrix Norms

In many applications it is appropriate to have some measure of the sizes or magnitude of vectors or matrices. This measure is called a norm and is denoted by \|.\|.

## Definition 4.2.1

Given a vector $\underline{x}$ then its norm $||\underline{x}||$ is a non-negative number with the following properties:
(1) $||\underline{x}|| \geqslant 0$, and $||\underline{x}||=0$ if and only if $\underline{x}=0$,
(2) $||\alpha \underline{x}||=|\alpha| \cdot| | \underline{x}| |$ for any complex scalar $\alpha$,
(3) $||\underline{x}+\underline{y}|| \leqslant||\underline{x}||+||\underline{y}||$ for vectors $\underline{x}$ and $\underline{y}$, and is known as the 'triangular inequality'.

From (3) we have,

$$
||\underline{x}-\mathbf{y}\|\geq 1| | \underline{x}| |-\| \underline{y}|| \mid .
$$

The most commonly used norms are the ones which are defined as follows: Definition 4.2.2

If $\underline{x}=\left(x_{1}, x_{2}, \ldots, x_{n}\right)^{T}$ is a $n$-dimensional vector then:

$$
\begin{aligned}
& \|\left.\underline{\underline{x}}\right|_{1}=\sum_{i=1}^{n}\left|x_{i}\right|, \\
& \left||\underline{x}|_{2}=\left(\sum_{i=1}^{n}\left|x_{i}\right|^{2}\right)^{\frac{1}{2}},\right. \text { which is known as the Euclidean norm, } \\
& ||\underline{x}||_{\infty}=\max \left|x_{i}\right| \text {, maximum or uniform norm. }
\end{aligned}
$$

The general norm case can be written for the above three special cases as:

$$
\left|\mid \underline{x} \|_{p}=\left(\sum_{i=1}^{n}\left|x_{i}\right|^{p}\right)^{1 / p} \quad p \geqslant 1 .\right.
$$

The matrix norm can be defined in a similar way.

## Definition 4.2.3

A norm of an $(n \times n)$ matrix, written as $||A||$, is a scalar satisfying the following properties:
(1) $||A|| \geqslant 0$ and $||A||=0$ if and only if $A=(0)$,
(2) $||\alpha \cdot A||=|\alpha| \cdot| | A| |$, for any scalar $\alpha$,
(3) $||A+B|| \leqslant||A||+||B||$, for matrices $A$ and $B$,
(4) $||A B|| \leqslant||A|| \cdot| | B| |$ for matrices $A$ and $B$.

In the same way as in vector norms,

$$
\begin{aligned}
& \|A\|_{1}=\max _{j} \sum_{i=1}^{n}\left|a_{i, j}\right| \text { (maximum absolute column sum), } \\
& \left.\|A\|_{2}=\text { (maximum eigenvalue of } A^{H} A\right)^{\frac{1}{2}}, \text { (spectral norm), } \\
& \|A\|_{\infty}=\max \sum_{j=1}^{n}\left|a_{i, j}\right| \text { (maximum absolute row sum). }
\end{aligned}
$$

Another norm which is compatible with the vector Euclidean norm is defined as follows:

$$
\|A\|_{E}=\left(\sum_{i, j}\left|a_{i, j}\right|^{2}\right)^{\frac{1}{2}}, \text { (Euclidean norm) }
$$

## Definition 4.2.4

A matrix norm $||A||$ is said to be compatible with a vector norm ||x|| if

$$
\| A \underline{x}| | \leqslant||A|| \cdot| | \underline{x}| | \text {, for all } x \neq 0 \text {. }
$$

## Definition 4.2.5

A matrix norm is said to be subordinate to the corresponding norm, if it can be constructed in the following form:

$$
\|A\|=\max _{\underline{x} \neq 0} \frac{\||\underline{\underline{x}}| \mid}{\|\underline{x}\|},
$$

or equivalently,

$$
\| A| |=\max _{\left||\underline{x}|_{1}\right.}| | A \underline{x}| |
$$

## Definition 4.2.6

Let $A$ be an $(n \times n)$ matrix with eigenvalues $\lambda_{i}, 1 \leqslant i \leqslant n$, then the spectral radius of $A$ can be defined as,

$$
\begin{equation*}
\rho(A)=\max _{1 \leqslant i \leqslant n}\left|\lambda_{i}\right| \tag{4.2.12}
\end{equation*}
$$

For any ( $n \times n$ ) matrix $A$ and any norm, it can be shown by definition (4.2.4),

$$
\begin{equation*}
\rho(A) \leqslant\|A\| \tag{4.2.13}
\end{equation*}
$$

Let $\lambda_{i}$ be an eigenvalue of $A$ and $x_{i}$ its corresponding eigenvector, then,

$$
A x_{i}=\lambda_{i} \underline{x}_{i}
$$

and

Thus,

$$
\lambda_{i} \leqslant\|A\|
$$

since $\lambda_{i}$ was arbitrarily chosen, then from the definition (4.2.6)

$$
\rho(A) \leqslant \| A| |
$$

$A$ sequence of matrices $A^{(1)}, A^{(2)}, A^{(3)}, \ldots$, of the same dimension is said to be convergent to a matrix $A$ (say) if and only if,

$$
\lim \left|\left|A-A^{(k)}\right|\right|=0
$$

## Definition 4.2.7

Let $A$ be a square matrix, then A converges to zero if the sequence of matrices $A, A^{2}, A^{3}, \ldots$ converges to the null matrix $O$ and is divergent otherwise.

Theorem 4.2.3
Let $A$ be an $(n \times n)$ matrix and if $||A||<l$ then

$$
\lim _{k \rightarrow \infty} A^{k}=0
$$

Proof
and since $||A||<1$ then $\lim _{k \rightarrow \infty} A^{k}=0$.

## Theorem 4.2.4

If $A$ is an ( $n \times n$ ) matrix, then $A$ is convergent if and only if $\rho(A)<l(\rho(A)$ is the spectral radius of the matrix $A)$.

## Proof

See Varga [1962], page 13.

## Property A and Consistent Ordering Matrices

The first definition was due to (Young, [1950]).
Definition 4.2.8 (Young (1971])
A matrix A of order $n$ has Property $A$ if there exists two disjoint subsets $S_{1}$ and $S_{2}$ of $W=\{1,2, \ldots, n\}$ (the set of the first $N$ positive integers), such that $S_{1}+S_{2}=W$ and such that if $i \neq j$ and if either $a_{i, j} \neq 0$ or $a_{j, i} \neq 0$, then $i \in s_{1}$ and $j \in s_{2}$ or else $i \in S_{2}$ and $j \in s_{1}$. If $s_{1}$ or $S_{2}$ is empty, then $A$ is, of course, diagonal.

If a matrix A has Property (A), it is always possible to rearrange the rows and corresponding columns of $A$, in order to obtain the matrix $\tilde{A}$ which has either the block tridiagonal form

or the form,

$$
\tilde{A}=\left[\begin{array}{ll}
D_{1} & F_{1}  \tag{4.2.15}\\
E_{1} & D_{2}
\end{array}\right]
$$

where $D_{i}$ are square diagonal matrices not necessarily of the same order. i.e. there exists a permutation matrix $P$ such that $\tilde{A}=P^{-1} A P$ has either the form ( 4.2 .14 ) or (4.2.15).

Definition 4.2.9 (Young [1971])
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 $i$.

### 4.3 BASIC METHODS FOR SOLVING A SET OF LINEAR EQUATIONS

A variety of real life problems may be solved by determining the solution of a set of simultaneous equations. We also find that, when we study the numerical methods of solving a boundary-value problem involving partial differential equations, then these problems also require the solution of sets of equations which are often very large in size.

A matrix-vector notation may be used to express the system of linear equations as,

$$
\begin{equation*}
A \underline{x}=\underline{b}, \tag{4.3.1}
\end{equation*}
$$

where $A$ is an ( $n \times n$ ) matrix of coefficients, $\underline{b}$ is a known $n$-vector and $\underline{x}$ is an unknown $n$-vector whose value is to be found. Provided that $\operatorname{det}(\mathrm{A})$ is non-zero, the unique solution of the equation is expressed simply as,

$$
\underline{x}=A^{-1} \underline{b},
$$

where $A^{-1}$ is the inverse of the matrix $A$.
The methods used to solve (4.3.1) can be classified into two classes, the class of direct methods (also known as the elimination methods) and the class of iterative methods (also known as the indirect methods).

### 4.3.1 Direct Methods

By a direct method we mean a method which calculates the required solution without any intermediate approximations. Direct methods are based ultimately on the process of the elimination of variables.

To solve equation (4.3.1) for the unknown n-vector $\underline{x}$, in the case when $A$ is upper-triangular with all diagonal elements non-zero. Then
the system of equations has the form,

$$
\begin{align*}
a_{1,1} x_{1}+a_{1,2} x_{2}+\ldots+a_{1, n-1} x_{n-1}+a_{1, n} x_{n} & =b_{1} \\
a_{2,2} x_{2}+\ldots+a_{2, n-1} x_{n-1}+a_{2, n} x_{n} & =b_{2} \\
\vdots & \vdots  \tag{4.3.2}\\
a_{n-1, n-1} x_{n-1}+a_{n-1, n} x_{n} & =b_{n-1} \\
a_{n, n} x_{n} & =b_{n}
\end{align*}
$$

The $n^{\text {th }}$ equation in the system (4.3.2) gives $x_{n}$ directly (since $a_{n, n} \neq 0$ ) i.e., $x_{n}=\frac{b_{n}}{a_{n, n}}$.

Since $x_{n}$ is known, then the $(n-1)$ th equation gives $x_{n-1}$ (since $a_{n-1, n-1} \neq 0$ and $x_{n}$ is now known), so

$$
\begin{equation*}
x_{n-1}=\frac{b_{n-1}^{-a_{n-1, n}} x_{n}}{a_{n-1, n-1}} \tag{4.3.3}
\end{equation*}
$$

and so on until finally the first equation gives $x_{1}$. In general, with $x_{k+1}, x_{k+2}, \ldots, x_{n}$ already computed, the $k^{\text {th }}$ equation can be uniquely solved for $x_{k}$, since $a_{k k} \neq 0$, to give,

$$
\begin{equation*}
x_{k}=\left(b_{k}-\sum_{j=k+1}^{n} a_{k, j} x_{j}\right) \cdot \frac{1}{\alpha_{k x}} \tag{4.3.4}
\end{equation*}
$$

This process of determining the solution of (4.3.2) is called backsubstitution. The amount of work involved in the back-substitution method is (Conte and Boor [1982]),
and

| $n$ | division |
| :--- | :--- |
| $\frac{1}{2} n(n-1)$ | multiplications |
| $\frac{1}{2} n(n-1)$ | additions |

If the coefficient matrix $A$ of the systen (4.3.1) is not uppertriangular, we have to reduce the system to an equivalent system with upper-triangular coefficient matrix. This latter system can then be
solved by back-substitution. This method is known as Gaussian Elimination. The elementary procedure in which the first equation is used to eliminate the first variable in the last ( $n-1$ ) equations, the new second equation is used to eliminate the second variable from the last ( $n-2$ ) equations, and so on. If ( $n-1$ ) such eliminations can be performed, then the resulting system is triangular and is easily solvable. A sequence of equivalent systems $A^{(k)} \underline{x}=\underline{b}^{(k)}, k=0,1,2, \ldots$, $n-1$, are derived from the given linear system $A \underline{x}=\underline{b}$ of order $n$. Here $A^{(0)} \underline{x=b}{ }^{(0)}$ is just the original system. The elimination process is carried out by each step as: if the coefficient $a_{k k}^{(k-1)}$ of $x_{k}$ in equation $k$ is not zero, then $\left(a_{i k}^{(k-1)} / a_{k k}^{(k-1)}\right.$ ) times equation $k$ is subtracted from equation $i$, thereby eliminating the unknown $x_{k}$ from equation $i$, $i=k+1$, $k+2, \ldots, n$ and $k=1,2, \ldots,(n-1)$. After $(n-1)$ steps of this procedure, one arrives at the system $A^{(n-1)}{\underline{x}=\underline{b}^{(n-1)} \text {, whose coefficient matrix is }}^{(n)}$ upper-triangular, so this system can now be solved by back-substitution.

The Gaussian elimination process is usually programmed to accommodate the coefficients and right hand side of the $n$ equations $A x=b$, in which case the storage required is $n(n+1)$ locations plus a possible further $n$ locations for the final solution vector. Finally, the amount of work involved, can be seen to be (Fox [1964]): ?
n
$\frac{1}{3} n^{3}+n^{2}-\frac{1}{3} n$
and
divisions
multiplications
additions .

In practice, however, especially in the solution of sets of
linear equations, this value of the computational work involved is never actually attained as the matrix A is never full. Instead A is in general a band matrix, i.e. $A \equiv\left(a_{i, j}\right)$ where

$$
a_{i, j}=0 \quad \left\lvert\, \begin{aligned}
& i-j>m \\
& j-i>m
\end{aligned}\right.
$$

so that the number of non-zero elements in each row is at most only $2 \mathrm{~m}+1$ the bandwidth of the matrix. In this case, the number of multiplications is of order $2 \mathrm{~m}^{2} \mathrm{n}$ in contrast to the factor $\mathrm{n}^{3} / 3$ for the full matrix (Martin [1966]).

Another method can be used to solve equation (4.3.1) which is known as LU Decomposition. The matrix A can be decomposed into a pair of factors $L$ and $U$, such that

$$
\begin{equation*}
A=L U, \tag{4.3.7}
\end{equation*}
$$

where $L$ is a unit lower triangular matrix and $U$ is an upper triangular matrix as shown below.

All the coefficients in $L$ and $U$ are initially unknown. The rule for matrix multiplication enables them to be found from the following equations:
$\left.\begin{array}{l}\text { for } j=i, i+1, \ldots, n, u_{i, j}=a_{i, j}-\sum_{k=1}^{i-1} \ell_{i, k} u_{k, j} \\ \text { for } j=i+1, i+2, \ldots, n, \ell_{j, i}=\frac{1}{u_{i, i}}\left[a_{j, i}-\sum_{k=1}^{i-1} \ell_{j, k} u_{k, i}\right]\end{array}\right\} \begin{aligned} & i=1,2, \ldots, n\end{aligned}$

This factorization method fails only if one of the diagonal elements $u$, which are used as divisors in the second equation of (4.3.8) proves to be zero.

Equation (4.3.1) can then be written as,

$$
\begin{equation*}
\text { LUX }=\underline{b} \text {, } \tag{4.3.9}
\end{equation*}
$$

and the solution of equation (4.3.1) is computed from equation (4.3.9) by introducing the column vector $y$ such that $\underline{y}=\underline{X}$ and then solving Ly=b for $\underline{y}$ by a forward substitution process and $U \underline{x}=\underline{y}$ for $\underline{x}$ by back substitution.

The amount of work needed to solve $A \underline{x}=\underline{b}$ using $L U$ decomposition requires (Vichnevetsky [1981]),

$$
\begin{array}{ll}
\mathrm{N}^{3} / 3+O\left(\mathrm{~N}^{2}\right) & \text { multiplications } \\
\mathrm{N}^{3} / 3+O\left(\mathrm{~N}^{2}\right) & \text { additions } \tag{4.3.10}
\end{array}
$$

As with all direct methods of solution, even if $A$ is in general a sparse matrix, $L$ and $U$ will be full (triangular) matrices. But if $A$ is banded, then the bandwidth of $L$ and $U$ will not exceed that of $A$. In many applications, if $A$ is not banded then we try to put it into band form by interchanging the appropriate columns and rows. Let $B$ be the bandwidth then

$$
a_{i, j}=0 \quad \text { for }|i-j|>B
$$

and those elements lying outside (2B+1) diagonal bands are zero as shown in Figure 4.2. In this case, the two matrices $L$ and $U$ formed by
the decomposition are also banded of width $B$ as shown in Figure 4.3. The work of decomposing $A$ into $L$ and $U$ is greatly reduced, i.e., it becomes linear in $n$ if $B \ll n$. By contrast, the $L U$ decomposition of a sparse, but not banded, matrix does not benefit from such an advantage, as shown in Figure 4.4 (Vichnevetsky [1981]).


FIGURE 4.2


FIGURE 4.3: Densities of $A^{-1}$ and of $L$ and $U$ for a banded sparse matrix


Both $L$ and $U$ full
FIGURE 4.4: Denseness of $A^{-1}$ and $L$ and $U$ for a general sparse matrix

Hence, to solve the system (4.3.1) in the banded case, by assuming $B \lll N$ requires,

$$
\begin{array}{ll}
N B^{2}+O(N B) & \text { multiplications } \\
N B^{2}+O(N B) & \text { additions } \tag{4.3.11}
\end{array}
$$

where $B$ is the bandwidth.

### 4.3.2 Iterative Methods

As opposed to the direct method of solving a set of linear equations by elimination, we now discuss iterative methods. Iterative methods are preferred over the direct methods when the coefficient matrix is sparse. They may be more economical for the core-storage requirements of a computer. They have the distinct advantage that they are selfcorrecting if an error is made, they may sometimes be used to reduce round-off error in the solutions computed by direct methods (i.e. iterative improvements).

In any iterative method, we begin with some initial approximation to the value of the variables. By substituting these into the righthand
sides of the set of equations generates new approximations which are usually closer to the true value if certain conditions with regard to the matrix are satisfied i.e. diagonal dominance. The new values are substituted into the righthand sides to generate a second approximation and the process is repeated until successive values of each of the variables are sufficiently allke to the specified or required number of decimal places. 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 increased.

So we can say that, given a non-singular system,

$$
\begin{equation*}
\mathrm{Ax}=\underline{\mathrm{b}} . \tag{4.3.12}
\end{equation*}
$$

and a sequence of approximate solution $\left\{\underline{x}^{(k)}\right\}$ such that $\underline{x}^{(k)} \rightarrow A^{-1} \underline{b}$ as $k \rightarrow \infty$.

Equation (4.3.12) can be rearranged by splitting the matrix $A$ in such a way that,

$$
A=D-L-U
$$

where $D$ is the diagonal matrix formed from the diagonal entries and $L$ and $U$ are the upper and lower triangular matrices respectively comprising of the similar entries in $A$. Equation (4.3.12) may now be written as,

$$
\begin{equation*}
(D-L-U) \underline{x}=\underline{b}, \tag{4.3.13}
\end{equation*}
$$

which can be written as,

$$
D \underline{x}=(L+U) \underline{x}+\underline{b}
$$

(i) The Jacobi Method or the method of Simultaneous Displacement. In the Jacobi iterative method the $(n+1)$ th iterative values are exclusively expressed in terms of the nth iterative values. Then the Jacobi method can be defined as,

$$
\begin{equation*}
D \underline{x}^{(k+1)}=(L+U) \underline{x}^{(k)}+\underline{b}, k, 0, \tag{4.3.14}
\end{equation*}
$$

by multiplying both sides by $D^{-1}$, the following equation is obtained,

$$
\begin{equation*}
\underline{x}^{(k+1)}=D^{-1}(L+U) \underline{x}^{(k)}+D^{-1} \underline{b} \tag{4.3.15}
\end{equation*}
$$

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

$$
\begin{equation*}
x_{i}^{(k+1)}=\frac{1}{a_{i i}}\left\{b_{i}+\sum_{j=1}^{i-1} a_{i, j} x_{j}^{(k)}+\sum_{j=i+1}^{n} a_{i, j} x_{j}^{(k)}\right\}, k \geqslant 0 \tag{4,3.16}
\end{equation*}
$$

In this method, the components of the vector $x^{(k)}$ must be saved while computing the components of $\underline{x}^{(k+1)}$.
(ii) The Gauss-Seidel Method (The GS Method) also known as the Successive Displacement method which converges more quickly (approximately twice) than the Jacobi method (Varga [1962]). In this method the new values $x_{i}^{(k+1)}$ are used as soon as they are available instead of $\mathrm{x}_{\mathrm{i}}^{(\mathrm{k})}$. The Gauss-Seidel iterative method can be defined by the equation,

$$
\begin{equation*}
D \underline{x}^{(k+1)}=L \underline{x}^{(k+1)}+\underline{u x}^{(k)}+\underline{b} \tag{4.3.17}
\end{equation*}
$$

giving,

$$
\begin{equation*}
(D-L) \underline{x}^{(n+1)}=U \underline{x}^{(n)}+\underline{b} \tag{4.3.18}
\end{equation*}
$$

which is written as,

$$
\begin{equation*}
\underline{x}^{(k+1)}=(D-L)^{-1} U \underline{x}^{(k)}+(D-L)^{-1} \underline{b} \tag{4.3.19}
\end{equation*}
$$

Since (D-L) is a non-singular matrix, equation (4.3.19) shows that the Gauss-Seidel point iteration matrix is (D-L) ${ }^{-1}$ U. From equation (4.3.17), the iteration of each point $x_{i}$ is given by,

$$
\begin{gather*}
x_{i}^{(k+1)}=\frac{1}{a_{i i}}\left\{b_{i}+\sum_{j=1}^{i} a_{i, j} x_{j}^{(k+1)}+\sum_{j=i+1}^{p} a_{i, j} x_{j}^{(k)}\right\}, \text { for } i=1,2, \\
\ldots, n \tag{4.3.20}
\end{gather*}
$$

The computational advantage of this iterative method is that it does not require the simultaneous storage of the two approximations $x_{i}^{(k+1)}$ and $x_{i}^{(k)}$ as in the point Jacobi iterative method.
(iii) The Successive Overrelaxation Method (S.O.R. Method). Related to the GS method is the S.O.R. method. In this method, the displacement or correction vector $\underline{d}^{(k)}=\underline{x}^{(k+1)}-\underline{x}^{(k)}$ of the S.O.R. method is taken to be a constant $\omega$ times the displacement vector $d_{1}^{(k)}$ defined by the GS iteration. Hence, from equation (4.3.17), the S.O.R. method is defined as,

$$
\begin{aligned}
D d_{1}^{(k)} & \left.=D \underline{x}^{(k+1)}-\underline{x}^{(k)}\right) \\
& =D \underline{x}^{(k+1)}-D \underline{x}^{(k)} \\
& =L \underline{x}^{(k+1)}+U \underline{x}^{(k)}-D \underline{x}^{(k)}+\underline{b}
\end{aligned}
$$

Thus, the S.O.R. iteration defined by,

$$
\underline{d}^{(k)}=\omega d_{1}^{(k)}
$$

can be written as,

$$
\underline{x}^{(k+1)}-\underline{x}^{(k)}=\omega D^{-1}\left(L \underline{x}^{(k+1)}+U \underline{x}^{(k)}+\underline{b}-D \underline{x}^{(k)}\right)
$$

therefore,

$$
\left(I-\omega D^{-1} L\right) \underline{x}^{(k+1)}=\left\{(1-\omega) I+\omega D^{-1} U\right\}_{\underline{x}}^{(k)}+\omega D^{-1} b,
$$

giving,

$$
\begin{equation*}
\underline{x}^{(k+1)}=\left(I-\omega D^{-1} L\right)^{-1}\left\{(1-\omega) I+\omega D^{-1} U\right\} \underline{x}^{(k)}+\left(I-\omega D^{-1} L\right) D^{-1} b . \tag{4.3.21}
\end{equation*}
$$

Therefore the point S.O.R. iteration matrix is

$$
\left(I-\omega D^{-1} L\right)^{-1}\left\{(1-\omega) I+\omega D^{-1} \tilde{U}\right\} .
$$

If $\omega=1$, the S.O.R. method reduces to that of Gauss-Seidel Method. The quantity $\omega$ is called an overrelaxation parameter, the choice of which determines the rapidity of convergence. From equation (4.3.21), we can
reformulate it in point form as,

$$
\begin{equation*}
x_{i}^{(k+1)}=\frac{\omega}{a_{i i}}\left\{b_{i}+\sum_{j=1}^{i-1} a_{i, j} x_{j}^{(k+1)}+\sum_{j=i+1}^{n} a_{i, j} x_{i}^{(k)}\right\}-(\omega-1) x_{i}^{(k)} \tag{4.3.22}
\end{equation*}
$$

### 4.4 CONVERGENCE OF POINT ITERATIVE METHODS

To find the conditions for the convergence of the iterative methods described in Section 4.3, let us consider the system of equations,

$$
\begin{equation*}
A \underline{x}=\underline{b} \text {, } \tag{4.4.1}
\end{equation*}
$$

where $A$ is an $(n \times n)$ matrix and $x$ and $\underline{b}$ are ( $n \times 1$ ) vectors. As shown in Section 4.3, the general form of a stationary linear iterative method may be written as,

$$
\begin{equation*}
\underline{x}^{(n+1)}=G \underline{x}^{(n)}+\underline{r} \tag{4.4.2}
\end{equation*}
$$

where $G$ is the corresponding iteration matrix for the specific method and $\underline{r}$ is a column vector of known values. $G$ and $\underline{r}$ are both defined as follows:-

$$
\begin{aligned}
& G= \begin{cases}D^{-1}(L+U) & \text { for Jacobi method } \\
(D-L)^{-1} U & \text { for Gauss-Seidel method } \\
\left(I-\omega D^{-1} L\right)^{-1}\left\{(1-\omega) I+\omega D^{-1} U\right\}, & \text { for S.O.R. method (usually }\end{cases} \\
& \underline{r}= \begin{cases}D^{-1} b & \text { denoted by } \left.L_{\omega}\right) . \\
(D-L)^{-1} \frac{b}{} \quad \text { for Jacobi method } \\
\left(I-\omega D^{-1} L\right)^{-1} D_{\omega D^{-1} \underline{b}} \quad \text { for Gauss-Seidel method }\end{cases}
\end{aligned}
$$

Equation (4.4.2) can be derived from equation (4.4.1) by rearranging them into the form,

$$
\begin{equation*}
\underline{x}=G \underline{x}+\underline{r}, \tag{4.4.3}
\end{equation*}
$$

i.e., the unique solution of $n$ Iinear equations $A \underline{x}=\underline{b}$ is the solution of equation (4.4.3). Alternatively, if we assume the iteration is convergent then, by (4.4.2),

$$
\lim _{n \rightarrow \infty} \underline{x}^{(n+1)}=\lim _{n \rightarrow \infty} \underline{x}^{(n)}=\underline{x}
$$

hence,

$$
\underline{x}=G \underline{x}+\underline{r} .
$$

Let the error at any stage be the difference between the true and approximate solutions, i.e.,

$$
\begin{equation*}
\underline{e}^{(n)}=\underline{x}^{(n)} \tag{4.4.4}
\end{equation*}
$$

then by subtracting equation (4.4.2) from equation (4.4.3) we have

$$
\begin{equation*}
e^{(n+1)}=G e^{(n)} \tag{4.4.5}
\end{equation*}
$$

Therefore,

$$
\begin{equation*}
\underline{e}^{(n)}=G \underline{e}^{(n-1)}=G^{2} e^{(n-2)}=\ldots=G^{(n)} e^{(0)}, \tag{4.4.6}
\end{equation*}
$$

where $\underline{e}^{(0)}={\underline{x}-\underline{x}^{(0)}}^{(0)}$ and $\underline{x}^{(0)}$ is a known set of initial values.
The sequence of iterative values $\underline{x}^{(1)}, \underline{x}^{(2)}, \ldots, x^{(n)}, \ldots$ will
converge to $x$ as $n$ tends to infinity if

$$
\begin{equation*}
\lim _{n \rightarrow \infty} e^{(n)}=\underline{o} \tag{4.4.7}
\end{equation*}
$$

From equation (4.4.7), this can happen if and only if $G^{(n)} \rightarrow(0)$ (the null matrix) as $k \rightarrow \infty$. By theorem (4.2.4), this will be true if and only if $\rho(G)<l$, which proves that iterative methods in the form of (4.4.2) converges if and only if $\rho(G)<1$.

## Corollary 4.4.1

A sufficient condition for the convergence of equation (4.4.2)

## is that

$$
\begin{equation*}
||G||<1 \tag{4.4.8}
\end{equation*}
$$

since $\rho(G) \leqslant||G|| \quad$ (from equation (4.2.13)).
In some cases it happens that $||G||>1$ but $\rho(G)<1$ which guarantees the convergence of the iteration process, thus it means this is a sufficient condition but not a necessary one.

### 4.5 RATE OF CONVERGENCE

In order to assess how iterative methods are effective, it is necessary to consider the number of iterations required for each convergence to a specified accuracy and the work done per each iteration. In practice, the usual approach is to iterate until the norm of the error vector $e^{(k)}$ is reduced to less than some predetermined value, say $\varepsilon$, of the norm of the initial vector $e^{(0)}$. From equation (4.4.6) we have,

$$
\begin{equation*}
\left\|\underline{e}^{(k)}\right\|=\left\|G_{e^{k}}^{(0)}\right\| \leqslant\left\|G^{k}\right\| \cdot\left\|e^{(0)}\right\| \tag{4.5.1}
\end{equation*}
$$

Then, if $\underline{e}^{(0)} \neq \underline{O}$,

$$
\begin{equation*}
\left\|\underline{e}^{(k)}\right\| /\left\|\underline{e}^{(0)}\right\| \leqslant\left\|G^{k}\right\| \tag{4.5.2}
\end{equation*}
$$

we require,

$$
\begin{equation*}
\left\|\underline{e}^{(k)}\right\| \leqslant \varepsilon\left\|\underline{e}^{(0)}\right\| \tag{4.5.3}
\end{equation*}
$$

where $\|$.$\| denotes \|.\|_{2}$ as defined in (4.2.2). From Section 4.4 we have $\lim _{k \rightarrow \infty}| | G^{k}| |=0$ if and only if $\rho(G)<1$ (also Young [1971], page 84). Equation (4.5.3) can be satisfied by choosing $k$ sufficiently large so that,

$$
\begin{equation*}
\left|\left|G^{k}\right|\right| \leqslant \varepsilon \tag{4.5.4}
\end{equation*}
$$

If $k$ is large enough so that $||G||<1$, it follows that equation (4.5.4) is equivalent to $;$

$$
\begin{equation*}
k \geqslant-\log \varepsilon /\left(-\frac{1}{k} \log | | G^{k}| |\right) \tag{4.5.5}
\end{equation*}
$$

and from this inequality a lower bound for the number of iterations for the iterative method can be obtained.

Young [1971] concluded that the average rate of convergence after $k$ iterations for any convergent iterative method in the form of equation
(4.4.2) is the quantity,

$$
\begin{equation*}
R_{k}(G)=\frac{-\log | | G^{k}| |}{k} \tag{4.5.6}
\end{equation*}
$$

If $R_{k}\left(G_{1}\right)<R_{k}\left(G_{2}\right)$, then $G_{2}$ is iteratively faster for $k$ iterations than $G_{1}$.

The asymptotic average rate of convergence is defined by,

$$
R(G)=\lim _{k \rightarrow \infty} R_{k}(G)=-\log \rho(G)
$$

Equation (4.5.7) holds, since,

$$
\begin{equation*}
\rho(G)=\lim _{k \rightarrow \infty}\left(| | G^{k}| |\right)^{1 / k} \tag{4.5.8}
\end{equation*}
$$

which is proved by Young [1971], page 87. $R(G)$ is referred to as the rate of convergence.

To obtain an estimate of the number of iterations, $k$, in equation (4.5.5) $[\rho(G)]^{k}$ is replaced instead by $\| G^{k}| |$ we see that $\varepsilon \approx[\rho(G)]^{k}$, and hence,

$$
\begin{equation*}
k \approx \frac{-\log \varepsilon}{-\log \rho(G)}=\frac{-\log \varepsilon}{R(G)} \tag{4.5.9}
\end{equation*}
$$

On the other hand, the value $k$ obtained from (4.5.9) could be very much lower when compared with the actual number required, in which $\left|\left|G^{k}\right|\right|$ will behave like $k \rho(G)^{k-1}$, rather than $\rho(G)^{k}$ (see young [1971]). In this case, the smallest value of $k$ such that,

$$
k[\rho(G)]^{k-1} \leqslant \varepsilon
$$

estimates the number of iterations required.

### 4.6 THE OPTIMUM ACCELERATION PARAMETER FOR THE SOR METHOD

In section 4.3 it was shown that the time taken for convergence using the S.O.R. method is less than that of the Jacobi or GaussSeidel methods. The convergence of the S.O.R. method depends on the value of the acceleration factor $\omega$ and ideally, we want the optimal value of $\omega$, say $\omega_{b}$, which minimises the spectral radius of the S.O.R. iterative matrix and thereby minimise the rate of convergence of the method. At the present time no formula exists for the determination of $u_{b}$ for an arbitrary set of linear equations, i.e. general A. However it can be calculated for many of the matrices $A$ derived from difference equations approximations to first- and second-order partial differential equations because their matrices are of a special type which possesses property (A), and the significance of this was first revealed by Young [1954]. Young proved that when a matrix possesses property (A) then it can be transformed into what he termed a consistently ordered matrix. Subject to this condition the eigenvalues of the S.O.R. iteration matrix $L_{\omega}$ associated with A are related to the eigenvalues $\mu$ of the corresponding point Jacobi iteration matrix $B$ associated with $A$ by the equation,

$$
\begin{equation*}
(\lambda+\omega-1)^{2}=\lambda \omega^{2} \mu^{2} \tag{4.6.1}
\end{equation*}
$$

From the above equation, it can be seen that,

$$
\begin{equation*}
\lambda^{\frac{1}{2}}=\frac{\omega \mu \pm \sqrt{\omega^{2} \mu^{2}-4(\omega-1)}}{2} \tag{4.6.2}
\end{equation*}
$$

Young [1954] shows that the rate of convergence is dependent on $\lambda$ and so to optimise the rate of convergence, $\bar{\lambda}$, the eigenvalue of maximum modulus of the S.O.R. iteration matrix $L_{\omega}$ must be minimised. This is achieved by making the square root in equation (4.6.2) equal to zero for $\bar{\mu}$, the spectral radius of the point Jacobi iteration matrix

B, i.e.,

$$
\begin{equation*}
\omega_{b}^{2} \bar{\mu}^{2}=4\left(\omega_{b}-1\right), 1 \leqslant \omega_{b} \leqslant 2, \tag{4.6.3}
\end{equation*}
$$

giving,

$$
\begin{equation*}
\omega_{b}=\frac{2}{1+\sqrt{1-\bar{\mu}^{2}}} \tag{4.6.4}
\end{equation*}
$$

is the value of $\omega$ which minimises $\rho\left(L_{\omega}\right)$, i.e., $\omega \neq \omega_{b}$, then

$$
\begin{equation*}
\rho\left(L_{\omega}\right)>\rho\left(L_{\omega_{\mathrm{b}}}\right) \tag{4.6.5}
\end{equation*}
$$

Using this optimum value of $\omega$, Young [1971] and Smith [1978] show that,

$$
\begin{equation*}
\bar{\lambda}=\rho\left(L_{\omega_{b}}\right)=\omega_{b}-1 . \tag{4.6.6}
\end{equation*}
$$

The eigenvalue of maximum modulus value of $G$ is called the spectral radius of $G$. As we know that the Gauss-Seidel method is the same as S.O.R. with $\omega=1$, and it can be shown that by substitution of this value for $\omega$ into (4.6.1), we have

$$
\begin{equation*}
\rho(G)=\rho(J)^{2} \tag{4.6.7}
\end{equation*}
$$

where $\rho(G)$ and $\rho(J)$ are the spectral radii of the Gauss-Seidel and point Jacobi iteration matrices respectively. Therefore, $\omega_{b}$ may be expressed in terms of $\rho(G)$, as

$$
\begin{equation*}
\omega_{b}=\frac{2}{1+\sqrt{1-\rho(G)}} \tag{4.6.8}
\end{equation*}
$$

The estimation of $\omega_{b}$ depends on whether $\rho(J)$ or $\rho(G)$ can be estimated. Several methods have been suggested by (Carre, [1961], Varga, [1962]) and Hageman and Kellogg [1968]), One which is the power method can be described as follows.

Assuming the matrix of the finite difference equations is consistently ordered and has property (A), calculate the sequence of
approximations $\underline{x}^{(1)}, \underline{x}^{(2)}, \ldots \underline{x}^{(k)}$, to the solution of the system of equations $A \underline{x}=\underline{b}$ by the Gauss-Seidel method and then we have,

$$
\begin{equation*}
\rho(G)=\lim _{k \rightarrow \infty} \frac{\left\|\underline{d}^{(k)}\right\|}{\left\|d^{(k-1)}\right\|} \tag{4.6.9}
\end{equation*}
$$

where ${\underset{d}{d}}^{(k)}$ is defined as,

$$
\begin{equation*}
\underline{d}^{(k)}=\underline{x}^{(k)}-\underline{x}^{(k-1)} \tag{4.6.10}
\end{equation*}
$$

and

$$
\begin{equation*}
\left|\left|\underline{d}^{(k)}\right|\right|=\left\{\sum_{j=1}^{n}\left(x_{j}^{(k)}-x_{j}^{(k-1)}\right)^{2}\right\}^{\frac{1}{2}} \tag{4.6.11}
\end{equation*}
$$

Thus, using the power method we can determine an approximate value of $\rho(G)$, which, in turn, can be substituted into equation (4.6.8) to give an estimate of the optimum acceleration factor, $\omega_{b}$.

Young [1971] and Smith [1978], show that the successive errors at any mesh point, after a large number of iterations, are related to the equation,

$$
\begin{equation*}
\left|e^{(n+1)}\right|=\rho\left|e^{(n)}\right| \tag{4.6.12}
\end{equation*}
$$

where $\rho$ is the spectral radius of $G$. Also, for theoretical purposes the asymptotic rate of convergence, $R$, is defined as,

$$
\begin{equation*}
R=-\log _{e}(\rho) \tag{4.6.13}
\end{equation*}
$$

They show that $R$ for the Gauss-Seidel method is twice that of the point Jacobi method, and $R$ for the S.O.R. method is approximately $2 / \varepsilon$ times that of Gauss-Seidel method, where $\mu^{2}=1-\varepsilon^{2}, \varepsilon$ being small for large $n$. The estimate of the number of iterations, $n$ that is necessary in order to make $e^{(n)}<\varepsilon$ may be obtained, where $\varepsilon$ is the required accuracy. From equation (4.6.9) it can be shown that (Young [1971]),

$$
\begin{equation*}
n \approx \frac{\log \varepsilon}{\log (\omega-1)} \tag{4.6.14}
\end{equation*}
$$

### 4.7 FINITE DIFFERENCE APPİOXIMATIONS TO DERIVATIVES

Many problems in engineering and science can be formulated in terms of partial differential equations or a set of such equations. Analytical or numerical methods may be used to find the solution to a partial differential equation, and approximation techniques may be used in both methods. Analytical approximation methods tend to be more difficult to apply than numerical methods. Of the numerical approximation methods available for solving differential equations those employing finite-differences are more frequently used and more universally applicable than any other.

Finite-difference methods are approximate in the sense that derivatives at a point are approximated by difference quotients over a small interval. Suppose we are given the interval [a,b], we divide the interval $[a, b]$ into $N$ equal parts of width $h$. We set $x_{0}=a, x_{N}=b$, and we define,

$$
x_{n}=x_{0}+n h, \quad n=1,2, \ldots, N-1
$$



Suppose $u$ and its derivatives are single-valued, finite and continuous functions of $x$, then by Taylor's theorem,

$$
\begin{equation*}
u(x+h)=u(x)+h u^{\prime}(x)+\frac{h^{2}}{2!} u^{\prime \prime}(x)+\frac{h^{3}}{3!} u^{\prime \prime \prime}(x)+\ldots \tag{4.7.1}
\end{equation*}
$$

and

$$
\begin{equation*}
u(x-h)=u(x)-h u^{\prime}(x)+\frac{h^{2}}{2!} u^{\prime \prime}(x)-\frac{h^{3}}{3!} u^{\prime \prime \prime}(x)+\ldots \tag{4.7.2}
\end{equation*}
$$

By adding (4.7.1) and (4.7.2) we get

$$
\begin{equation*}
u(x+h)+u(x-h)=2 u(x)+h^{2} u^{\prime \prime}(x)+O\left(h^{4}\right) \tag{4.7.3}
\end{equation*}
$$

where $O\left(h^{4}\right)$ denotes terms containing fourth and higher power of $h$.

Assuming these are negligible in comparison with the lower powers of h it follows that,

$$
\begin{equation*}
u^{\prime \prime}(x)=\frac{d^{2} u}{d x^{2}}=\frac{1}{h^{2}}\{u(x+h)-2 u(x)+u(x-h)\}+o\left(h^{2}\right) \tag{4.7.4}
\end{equation*}
$$

with the leading error on the right-hand side of order $h^{2}$. Subtracting equation (4.7.2) from equation (4.7.1) and neglecting the terms of order $h^{3}$ leads to

$$
\begin{equation*}
u^{\prime}(x)=\frac{d u}{d x}=\frac{1}{2 h}[u(x+h)-u(x-h)]+o\left(h^{2}\right) \tag{4.7.5}
\end{equation*}
$$

with an error of order $h^{2}$.
As shown in Figure 4.2, Equation (4.7.5) clearly approximates the slope of the tangent at $p$ by the slope of the chord $A B$, and is called a central-difference approximation. We can also approximate the slope of the tangent at $P$ by either the slope of the chord $P B$, giving the forwarddifference formula,

$$
\begin{equation*}
u^{\prime}(x) \cong \frac{1}{h}\{u(x+h)-u(x)\} \tag{4.7.6}
\end{equation*}
$$

or the slope of the chord AP giving the backward-difference formula,

$$
\begin{equation*}
u^{\prime}(x) \cong \frac{1}{h}\{u(x)-u(x-h)\} \tag{4.7.7}
\end{equation*}
$$



FIGURE 4.2

By assuming second and higher powers of $h$ are negligible, both equations (4.7.6) and (4.7.7) can be obtained from equations (4.7.1) and (4.7.2) respectively. This shows that the leading errors in these forward and backward-difference formula are both of order $h$.

To solve a boundary-value problem by the method of finite difference, every derivative appearing in the equation, as well as in the boundary conditions, is replaced by an appropriate finite difference approximation.

In our model problem, we consider the two-dimensional problem for the Laplace equation,

$$
\begin{equation*}
\frac{\partial^{2} u}{\partial x^{2}}+\frac{\partial^{2} u}{\partial y^{2}}=0 \tag{4.7.8}
\end{equation*}
$$

with dependent variable $u$ applied to a connected region $R$ in the $x-y$ plane. Equation (4.7.8) mathematically represents a problem with temperatures known on each boundary and is said to have Dirichlet boundary conditions. We require to determine the solution $u(x, y)$ inside the region $R$. We will refer to our model throughout the thesis as the Dirichlet problem.

Now we subdivide the $x-y$ plane into sets of equal rectangles as shown in Figure 4.3. The points of intersection of $x$ lines and $y$ lines are known as the mesh points. Replacing the derivatives by difference quotients which approximate the derivatives at each point of the mesh, we get a set of $M$ equations with $M$ unknowns. Therefore, the solution of the $M$ equations yields an approximation to the partial differential equation.

Consider now the uniform grid mesh of size $h$ on the unit square as shown in Figure 4.3.


FIGURE 4.3

Assuming that $u(x, y)$ is differentiable, then by Taylor's theorem we have,

$$
\begin{align*}
& u(x \pm h, y)= u(x, y) \pm h \frac{\partial u}{\partial x}+\frac{h^{2}}{2!} \frac{\partial^{2} u}{\partial x^{2}} \pm \frac{h^{3}}{3!} \frac{\partial^{3} u}{\partial x^{3}}+\frac{h^{4}}{4!} \frac{\partial^{4} u}{\partial x^{4}} \pm \ldots \text { (4.7.9) } \\
& u(x, y \pm h)= u(x, y) \pm h \frac{\partial u}{\partial y}+\frac{h^{2}}{2!} \frac{\partial^{2} u}{\partial y^{2}} \pm \frac{h^{3}}{3!} \frac{\partial^{3} u}{\partial y^{3}}+\frac{h^{4}}{4!} \frac{\partial^{4} u}{\partial x^{4}} \pm \ldots \text { (4.7.10) }  \tag{4.7.10}\\
& u(x+h, y \pm h)= u(x, y)+h\left(\frac{\partial u}{\partial x} \pm \frac{\partial u}{\partial y}\right)+\frac{h^{2}}{2!}\left(\frac{\partial^{2} u}{\partial x^{2}} \pm 2 \frac{\partial^{2} u}{\partial x \partial y}+\frac{\partial^{2} u}{\partial y^{2}}\right) \\
&+\frac{h^{3}}{3!}\left(\frac{\partial^{3} u}{\partial x^{3}} \pm 3 \frac{\partial^{3} u}{\partial x^{2} \partial y}+3 \frac{\partial^{3} u}{\partial x \partial y^{2}} \pm \frac{\partial^{3} u}{\partial y^{3}}\right)+\frac{h^{4}}{4!}\left(\frac{\partial^{4} u}{\partial x^{4}} \pm\right. \\
&\left.4 \frac{\partial^{4} u}{\partial x^{3} \partial y}+6 \frac{\partial^{4} u}{\partial x^{2} \partial y^{2}} \pm 4 \frac{\partial^{4} u}{\partial x \partial y^{3}}+\frac{\partial^{4} u}{\partial y^{4}}\right)+\ldots \tag{4.7.11}
\end{align*}
$$

$$
\begin{align*}
u(x-h, y \pm h)= & u(x, y)-h\left(\frac{\partial u}{\partial x} \pm \frac{\partial u}{\partial y}\right)+\frac{h^{2}}{2!}\left(\frac{\partial^{2} u}{\partial x^{2}} \pm 2 \frac{\partial^{2} u}{\partial x \partial y}+\frac{\partial^{2} u}{\partial y^{2}}\right)-\frac{h^{3}}{3!} \\
& \left(\frac{\partial^{3} u}{\partial x^{3}} \pm 3 \frac{\partial^{3} u}{\partial x^{2} \partial y}+3 \frac{\partial^{3} u}{\partial x \partial y^{2}} \pm \frac{\partial^{3} u}{\partial y^{3}}\right)+\frac{h^{4}}{4!}\left(\frac{\partial^{4} u}{\partial x^{4}} \pm 4 \frac{\partial^{4} u}{\partial x^{3} \partial y}\right. \\
& \left.+6 \frac{\partial^{4} u}{\partial x^{2} \partial y^{2}} \pm 4 \frac{\partial^{4} u}{\partial x \partial y^{3}}+\frac{\partial^{4} u}{\partial y^{4}}\right) \tag{4.7.12}
\end{align*}
$$

where the points ( $x \pm h, y$ ), $(x, y \pm h)$ and ( $x \pm h, y \pm h$ ) are contained in $R$.
Equations (4.7.9) and (4.7.10) respectively give,

$$
\begin{align*}
\frac{\partial u}{\partial x} & =\frac{u(x+h, y)-u(x, y)}{h}+o(h),  \tag{4.7.13}\\
& =\frac{u(x, y)-u(x-h, y)}{h}+o(h)  \tag{4.7.14}\\
\frac{\partial u}{\partial y} & =\frac{u(x, y+h)-u(x, y)}{h}+o(h),  \tag{4.7.15}\\
& =\frac{u(x, y)-u(x, y-h)}{h}+O(h) \tag{4.7.16}
\end{align*}
$$

More accurate approximations are,

$$
\begin{aligned}
& \frac{\partial u}{\partial x}=\frac{u(x+h, y)-u(x-h, y)}{2 h}+o\left(h^{2}\right), \\
& \frac{\partial u}{\partial y}=\frac{u(x, y+h)-u(x, y-h)}{2 h}+o\left(h^{2}\right) .
\end{aligned}
$$

A combination of equations (4.7.9) and (4.7.10) give approximations to the second-order derivatives

$$
\begin{align*}
& \frac{\partial^{2} u}{\partial x^{2}}=\frac{u(x+h, y)-2 u(x, y)+u(x-h, y)}{h^{2}}+o\left(h^{2}\right)  \tag{4.7.17}\\
& \frac{\partial^{2} u}{\partial y^{2}}=\frac{u(x, y+h)-2 u(x, y)+u(x, y-h)}{h^{2}}+o\left(h^{2}\right) \tag{4.7.18}
\end{align*}
$$

A combination of equations (4.7.11) and (4.7.12) gives an approximation to the second-order mixed derivative

$$
\begin{equation*}
\frac{\partial^{2} u}{\partial x \partial y}=\frac{u(x+h, y+h)-u(x+h, y-h)-u(x-h, y+h)+u(x-h, y-h)}{4 h^{2}} \tag{4.7.19}
\end{equation*}
$$

The mesh point $\left(x_{i}, y_{i}\right)=(i h, j h)$, let $u\left(x_{i}, y_{i}\right)$ be $u_{i, j}$, then
Laplace's equation

$$
\frac{\partial^{2} u}{\partial x^{2}}+\frac{\partial^{2} u}{\partial y^{2}}=0
$$

can be replaced at the point $\left(X_{i}, y_{j}\right)$ by the finite difference equation which is obtained from adding equations (4.7.17) and (4.7.18), thus,

$$
\begin{equation*}
\frac{1}{h^{2}}\left(u_{i+1, j}+u_{i-1, j}+u_{i, j+1}+u_{i, j-1}-4 u_{i, j}\right)=0 \tag{4.7.20}
\end{equation*}
$$

which can be written as,

$$
\begin{equation*}
4 u_{i, j} u_{i+1, j}^{-u_{i-1, j}} \mathrm{u}_{i, j+1}^{-u_{i, j-1}}=0 \tag{4.7.21}
\end{equation*}
$$

which is known as the 5-point finite difference equation.
A set of simultaneous equations may be obtained from equation (4.7.20) and whose solution is a finite-difference approximation of the exact solution $\left\{u_{i, j}\right\}$ at the internal mesh points. On the other hand, the computational molecule in Figure 4.4 corresponds to equation (4.7.20).


FIGURE 4.4: Five-point computational molecule for the Laplace operator

As an example, the ( $16 \times 16$ ) matrix illustrated in Figure 4.5
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, therefore,


FIGURE 4.5

Figure 4.6 shows the $(16 \times 16)$ matrix which is obtained using the computational molecule in Figure 4.3.


In Figure 4.5, at most five non-zero entries are contained in each row, i.e. the matrix is sparse so that the computation of any iterative method concentrates only on these five points in each row.

### 4.8 BLOCK ITERATIVE METHODS

The iterative methods for solving a system of linear equations described in Section 4.3 belongs to a class known as point iterative methods, that is, at any one time only a single equation of the system is treated. An extension of these methods are the block (group) iterative methods. The principle inherent'in the group iterative method is to group a certain number of individual equations (mesh points) and treat this group similar to the way a single point is treated in the point iterative method.

Consider a system of linear equations,

$$
\begin{equation*}
\sum_{j=1}^{n} a_{i, j} x_{j}=b_{i}, i=1,2, \ldots, n \tag{4.8.1}
\end{equation*}
$$

which can be written in matrix form as,

$$
\begin{equation*}
\mathrm{Ax}=\underline{\mathrm{b}} . \tag{4.8.2}
\end{equation*}
$$

In the group iterative methods the equations in (4.8.1) are partitioned into different groups numbered $1,2, \ldots, N$, such that each of the above equations belong to one and only one group. Then, the corresponding unknowns $X_{i}$ of each group are solved in which the other unknowns belonging to the remaining groups are treated as known quantities. In this case the equations for $i=1,2, \ldots, n_{1}$ constitute the first group, those for $i=n_{1}+1, n_{1}+2, \ldots, n_{2}$, constitute the second group and, in general, the equations for $i=n_{r}+1, n_{r}+2, \ldots, n_{r+1}$ constitute the $(r+1)$ st group and $n_{N}=N$ where $N$ represents the number of distinct groups.

In order to construct a group iterative method to solve equation (4.8.2), we first divide the integers $1,2, \ldots, n$ into $N \leqslant n$ distinct sets such that each integer belongs to one and only one set. Note that it
is not necessary for the groups to consist of consecutive integers although the groups are ordered by the following definitions.

Definition 4.8.1 (Young, [1971])
An ordered grouping $\pi$ of $T=\{1,2, \ldots, n\}$ (the set of the first $n$ positive integers), is a subdivision of $T$ into disjoint subsets $G_{1}, G_{2}$, $\ldots, G_{N}$ such that $G_{1} \cup G_{2} \cup \ldots \cup G_{N}=T$.

Two grouping $\pi$ and $\pi^{\prime}$ defined by $G_{1}, G_{2}, \ldots, G_{N}$ and $G_{1}, G_{2}, \ldots, G_{N}$ ', respectively, are identical if $N=N$, and if $G_{1}=G_{1}, G_{2}=G_{2}, \ldots, G_{N}=G_{N}$, . As an example, for $n=5$, we have the following ordered groupings:

$$
\begin{aligned}
& \pi_{0}: G_{1}=\{1\}, \quad G_{2}=\{2\}, \quad G_{3}=\{3\}, \quad G_{4}=\{4\}, \quad G_{5}=\{5\} \\
& \pi_{1}: G_{1}=\{1,2\}, \quad G_{2}=\{3,4\}, \quad G_{3}=\{5\} \\
& \pi_{2}: G_{1}=\{1,5\}, \quad G_{2}=\{2,4\}, \quad G_{3}=\{3\} \\
& \pi_{3}: G_{1}=\{1,3,5\}, G_{2}=\{2,4\}, \\
& \pi_{4}: G_{1}=\{3\}, G_{2}=\{1,2,4,5\} \\
& \pi_{5}: G_{1}=\{4\}, \quad G_{2}=\{1,2,5\}, G_{3}=\{3\}, \text { etc. }
\end{aligned}
$$

It is clear that $\pi_{0}$ and $\pi_{1}$ constitute partitionings.
Now, let us apply the above definition to the linear system (4.8.2) and let $\pi$ be an ordered grouping of equations in that system. We define the submatrices $A_{k, \ell}$, for $k, \ell=1,2, \ldots, N$, such that $A_{k, \ell}$, is formed from the matrix A by deleting all rows except those corresponding to $G_{k}$ and all columns except those corresponding to $G_{\ell}$. We also define the vectors $X_{k}$ and $B_{k}$, for $k=1,2, \ldots, N$, such that each $X_{k}$ and $B_{k}$ are formed from vector $x$ and $\underline{b}$ respectively by deleting all elements except those corresponding to group $G_{k}$. Equation (4.8.2) can be written as,

$$
\begin{equation*}
\sum_{\ell=1}^{N} A_{k, \ell} X_{\ell}=B_{k} \quad, \quad k=1,2, \ldots, N . \tag{4.8.3}
\end{equation*}
$$

As an example, if $\mathrm{n}=5$ and the ordered grouping is defined by $G_{1}=\{1,3,5\}, G_{2}=\{2,4\}$, we have,

$$
\begin{aligned}
& A_{11}=\left[\begin{array}{lll}
a_{1,1} & a_{1,3} & a_{1,5} \\
a_{3,1} & a_{3,3} & a_{3,5} \\
a_{5,1} & a_{5,3} & a_{5,5}
\end{array}\right], \quad A_{12}=\left[\begin{array}{ll}
a_{1,2} & a_{1,4} \\
a_{3,2} & a_{3,4} \\
a_{5,2} & a_{5,4}
\end{array}\right], \\
& x_{1}=\left[\begin{array}{l}
x_{1} \\
x_{3} \\
x_{5}
\end{array}\right], \quad B_{1}=\left[\begin{array}{l}
b_{1} \\
b_{3} \\
b_{5}
\end{array}\right], \quad A_{21}=\left[\begin{array}{lll}
a_{2,1} & a_{2,3} & a_{2,5} \\
a_{4,1} & a_{4,3} & a_{4,5}
\end{array}\right] \text {, } \\
& A_{22}=\left[\begin{array}{ll}
a_{2,2} & a_{2,4} \\
a_{4,2} & a_{4,4}
\end{array}\right], \quad x_{2}=\left[\begin{array}{l}
x_{2} \\
x_{4}
\end{array}\right], B_{2}=\left[\begin{array}{l}
b_{2} \\
b_{4}
\end{array}\right] .
\end{aligned}
$$

Equation (4.8.3) becomes,

$$
\begin{array}{ll}
{\left[\begin{array}{lll}
a_{1,1} & a_{1,3} & a_{1,5} \\
a_{3,1} & a_{3,3} & a_{3,5} \\
a_{5,1} & a_{5,3} & a_{5,5}
\end{array}\right]} & {\left[\begin{array}{l}
x_{1} \\
x_{2} \\
x_{3}
\end{array}\right]+\left[\begin{array}{ll}
a_{1,2} & a_{1,4} \\
a_{3,2} & a_{3,4} \\
a_{5,2} & a_{5,4}
\end{array}\right]}
\end{array}\left[\begin{array}{l}
x_{2} \\
{\left[\begin{array}{lll}
a_{2,1} & a_{2,3} & a_{2,5} \\
a_{4,1} & a_{4,3} & a_{4,5}
\end{array}\right]=\left[\begin{array}{l}
b_{1} \\
b_{3} \\
b_{5}
\end{array}\right]}
\end{array}\left[\begin{array}{l}
x_{1} \\
x_{3} \\
x_{5}
\end{array}\right]+\left[\begin{array}{ll}
a_{2,2} & a_{2,4} \\
a_{4,2} & a_{4,4}
\end{array}\right] \quad\left[\begin{array}{l}
x_{2} \\
x_{4}
\end{array}\right]=\left[\begin{array}{l}
b_{2} \\
b_{4}
\end{array}\right]\right.
$$

The matrix A will be partitioned into blocks (groups) according to $\pi$ and take the form, -

$$
A=\left[\begin{array}{llll}
A_{1,1} & A_{1,2} & \cdots \cdots \cdots & A_{1, N}  \tag{4.8.4}\\
A_{2,1} & A_{2,2} & \cdots \cdots \cdots & A_{2, N} \\
1 & & & \\
1 & & \cdots & A_{N, N}
\end{array}\right]
$$

where $N \leqslant n$ and $A_{k, k}, k=1,2, \ldots, N$ are square matrices and non-singular.

From this partitioning of the matrix $A$, we define the matrices,

and

where $D$ is a block diagonal matrix and the matrices $E$ and $F$ are strictly lower and upper block triangular matrix respectively, and

$$
\begin{equation*}
A=D-E-F \tag{4.8.6}
\end{equation*}
$$

Assuming that all submatrices $A_{i, i}$ are non-singular, the various block iterative schemes can now be defined as follows.

The block Jacobi iterative method is defined by,

$$
\begin{equation*}
A_{k, k} x_{k}^{(n+1)}=-\sum_{\substack{\ell=1 \\ \ell \neq k}}^{N} A_{k}, \ell_{\ell}^{(n)}+v_{k}, k=1,2, \ldots, N \tag{4.8.7}
\end{equation*}
$$

or equivalently,

$$
\begin{equation*}
x_{k}^{(n+1)}=\sum_{\substack{\ell=1 \\ \ell \neq k}}^{N} B_{k, \ell} X_{\ell}^{(n)}+C_{k}, \quad k=1,2, \ldots, N \tag{4.8.8}
\end{equation*}
$$

where,
and

$$
B_{k, \ell}= \begin{cases}-A_{k, k}^{-1} A_{k, \ell} & \text { if } k \neq \ell \\ 0 & \text { if } k=\ell\end{cases}
$$

$$
C_{k}=A_{k, k}^{-1} \cdot V_{k}
$$

We may write (4.8.8) in the matrix form,

$$
\underline{x}^{(n+1)}=B^{(\pi)} \underline{x}^{(n)}+\underline{C}^{(\pi)}
$$

where,

$$
\mathrm{B}^{(\pi)}=\left(\mathrm{D}^{(\pi)}\right)^{-1}\left(\mathrm{E}^{(\pi)}+\mathrm{F}^{(\pi)}\right) \text {, The block Jacobi matrix (4.8.11) }
$$

and

$$
\begin{align*}
& \underline{C}^{(\pi)}={\left(D^{(\pi)}\right.}^{(\pi-1},  \tag{4.8.12}\\
& D^{(\pi)}=\operatorname{diag} A \tag{4.8.13}
\end{align*}
$$

$E^{(\pi)}$ and $F^{(\pi)}$ are again strictly lower and upper triangular matrices.
For block Gauss-Seidel iterative method we have,

$$
\begin{equation*}
A_{k, k} X_{k}^{(n+1)}=-\sum_{\ell=1}^{k-1} A_{k, \ell} X_{\ell}^{(n+1)}-\sum_{\ell=k+1}^{N} A_{k, \ell^{\prime}} X_{\ell}^{(n)}+V_{k}, k=1,2, \ldots, N \tag{4.8.14}
\end{equation*}
$$

or

$$
\begin{equation*}
x_{k}^{(n+1)}=\sum_{\ell=1}^{k-1} B_{k, \ell} x_{\ell}^{(n+1)}+\sum_{\ell=k+1}^{N} B_{k, \ell} X_{\ell}^{(n)}+C_{k}, k=1,2, \ldots, N, \tag{4.8.15}
\end{equation*}
$$

where $B_{k, \ell}$ and $C_{k}$ are as given in (4.8.9). This can also be written in the matrix form,

$$
\begin{equation*}
\underline{x}^{(n+1)}=L^{(\pi)} \underline{x}^{(n)}+\left(D^{(\pi)}-E^{(\pi)}\right)^{-I_{b}} . \tag{4,8.16}
\end{equation*}
$$

where,

$$
\begin{equation*}
L^{(\pi)}=\left(\mathrm{D}^{(\pi)}-\mathrm{E}^{(\pi)}\right)^{-1} \mathrm{~F}^{(\pi)} \text {, The block Gauss-Seidel matrix. } \tag{4.8.17}
\end{equation*}
$$

For the Block S.O.R. (BSOR) iterative method we have,

$$
\begin{equation*}
\underline{x}^{(n+1)}=L_{\omega}^{(\pi)} \underline{x}^{(n)}+\omega\left(I-\omega L{ }^{(\pi)} \underline{c}_{\underline{c}}(\pi),\right. \tag{4.8.18}
\end{equation*}
$$

where,

$$
\begin{align*}
& L_{\omega}^{(\pi)}=\left(I-\omega L^{(\pi)}\right)^{-1}\left(\omega \mathrm{R}^{(\pi)}+(1-\omega) I\right) \text { The B.S.O.R. iteration matrix. }  \tag{4.8.19}\\
& L^{(\pi)}=\left(\mathrm{D}^{(\pi)}\right)^{-1} E^{(\pi)},  \tag{4.8.20}\\
& R^{(\pi)}=\left(D^{(\pi)}\right)^{-1} \mathrm{~F}^{(\pi)} . \tag{4.8.21}
\end{align*}
$$

For the analysis and convergence of the methods we follow the following definitions and theorems, which is a generalization of Young's definition of Property (A).

First we define the $(\ell \times \ell)$ matrix $z=\left(z_{i, j}\right)$ by,
and

$$
\left.\begin{array}{l}
z_{i, j}=1, \quad \text { if } A_{i, j} \neq 0  \tag{4,8.22}\\
z_{i, j}=0, \\
\text { if } A_{i, j}=0
\end{array}\right\}
$$

where the matrix $A$ and an ordered grouping $\pi$, with $\&$ groups are given.

## Definition 4.8.2

The matrix A has Property $A^{(\pi)}$, if $Z$ has Property A.

## Definition 4.8.3

The matrix A is a $\pi$-consistently ordered matrix if $Z$ is consistently ordered.

Definition 4.8.4 (Arms, Gates and Zondek [1956])
The matrix A has Property $A^{\pi}$ for a given partition $\pi$ if there exist two disjoint subsets $S$ and $T$ of $W$, the set of first $N$ positive integers, such that $S \cup T=W$ and such that if $A_{k, \ell} \neq 0$. Then, either $k=\ell$ or $K \in S$ and $\ell \in T$ or $k \in T$ and $\ell \in S$.

The above definition is a generalization of Young's Property $A^{(\pi)}$ definition.

## Definition 4.8.4

An ordering $\ell$-tuple for $A$ will be an $\ell$-tuple $v^{\pi}=\left(v_{1}^{\pi}, v_{2}^{\pi}, \ldots, v_{\ell}^{\pi}\right)$, where each $v_{s}^{\pi}$ is an integer, such that, if $A_{i, j} \neq O$ and $i \neq j$, then $\left|v_{i}^{\pi}-v_{j}^{\pi}\right|=1$.

## Theorem 4.8.1

> A matrix A has Property $A^{(\pi)}$ if and only if there exists an ordering l-tuple for $A$.

## Proof:

See Young [1971], page 148.

Theorem 4.8.2
If $A$ is a symmetric matrix and $D^{(\pi)}$ is positive definite, then $\rho\left(L_{\omega}^{(\pi)}\right)<1$ if and only if $A$ is positive definite and $0<\omega<2$.
proof:
See Young [1971], page 463.

For a symmetric matrix with $D^{(\pi)}$ positive definite, Theorem (4.8.2) shows the B.S.O.R. method converges. Now we try to find whether there is an optimal $\omega$.

Theorem 4.8.3
If A has property $A^{(\pi)}$ and is consistently ordered, with $0<\omega<2$, and if $\lambda$ is a non-zero eigenvalue of $L_{\omega}^{(\pi)}$, and if $\mu$ satisfies,

$$
\begin{equation*}
(\lambda-\omega-1)^{2}=\lambda \omega^{2} \mu^{2} \tag{4.8.23}
\end{equation*}
$$

then $\mu$ is an eigenvalue of $B^{(\pi)}$. Conversely, if $\mu$ is an eigenvalue of $B^{(\pi)}$ and if $\lambda$ satisfies (4.8.23), then $\lambda$ is an eigenvalue of $L_{\omega}^{(\pi)}$.

From equation (4.8.23) the optimal relaxation factor $\omega_{b}$ can be found in terms of $\vec{\mu}$, where $\bar{\mu}$ is the spectral radius of $B^{(\pi)}$ and the matrix $A$ is symmetric, positive definite and has property $A^{(\pi)}$. The value $\omega_{\mathrm{b}}$ is optimal in the sense that the spectral radius $\bar{\lambda}$ of $L_{\omega}^{(\pi)}$ is minimal so that the convergence rate is greatest. The relations are given by,

$$
\begin{equation*}
\omega_{\mathrm{b}}=\frac{2}{1+\sqrt{1-\mu^{2}}} \tag{4.8.24}
\end{equation*}
$$

and $\quad \bar{\lambda}=\rho\left(L_{\omega}^{(\pi)}\right)=\omega_{b}-1$.

Further we have,

$$
\begin{equation*}
\rho\left(L_{\omega_{b}}^{(\pi)}\right)<\rho\left(L_{\omega}^{(\pi)}\right), \omega \neq \omega_{b} \tag{4.8.26}
\end{equation*}
$$

and, asymptotically, as $\bar{\mu}+1$, we obtained the relation,

$$
\begin{equation*}
\left.\mathrm{R}\left(L_{\omega_{\mathrm{b}}}^{(\pi)}\right) \approx 2 \sqrt{\mathrm{R}\left(L^{\pi}\right)}=2 \sqrt{2 \mathrm{R}(\mathrm{~B}}{ }^{(\pi)}\right) \tag{4.8.26}
\end{equation*}
$$

As an example, in our model problem if all the points in the blocks are on two columns (or rows), or on three columns (or rows) then methods based on the use of such blocks are called two-line iterative methods and three-line iterative methods respectively as shown in Figure $4.7 \mathrm{a}, \mathrm{b}$.

a) Two-line block


FIGURE 4.7: Grouping for the $52 L O R$ and $S 3 L O R$ method.

### 4.9 PARALLEL ITERATIVE METHODS

Parallel iterative algorithms can be classified into two classes known as the class of "synchronised algorithms", or roughly, parallel algorithms for SIMD machines (see, for example, the surveys by Miranker [1971] and by Heller [1978]). The second class is known as the class of "asynchronous algorithms". In a synchronised iterative algorithm, the iterative function (a task) is decomposed into subtasks so that at each iterative step, the subtask is solved by one process of the algorithm. The processes are synchronised at the end of each iteration (which are the interaction points). At these points the processes may be blocked while waiting for inputs, so the performance of the algorithm is degraded. The performance degradation expected increases as the number of synchronised processes increases. Asynchronous parallel algorithms arise naturally in the use of multiprocessors, where the processors are not synchronised and the communication between cooperating processors is by means of shared data. When the fluctuations in the computation time are large, asynchronous algorithms are in general more efficient than synchronised ones for the following reasons. First, the processes never waste any time in waiting for inputs. Second, the algorithm can take advantage of processes which run fast. Results produced by those processes can be immediately used. Third, the algorithms are "adaptive", so the processes can finish about the same time.

It was argued that the original form of the traditional iterative methods that was discussed in Section 4.3 are not suitable for implementation on parallel computers since they require some form of synchronisation. On the other hand, many improvements have been studied
by different researchers to improve the above arguments.
Stone [1973] in his method to solve tridiagonal systems of linear equations using the ILLIAC IV computer (SIMD-type computer) which performs $N$ simultaneous computations where $N=64,128,256$ or 512 , a result N times faster than the serial computer of the same inherent speed was expected. Actually, inefficiencies due to overhead and constraints on data communication among processors will reduce the speed increase to $K N$ where $0 \leqslant k \leqslant l$. The parallel algorithm presented by Stone is based upon the LU decomposition, i.e., the matrix of coefficients decomposes into upper and lower triangular matrices. By using ILLIAC IV, each processor is assigned to each component of the known vector. The processors all work simultaneously, therefore, data can be communicated among the processors in one of two ways. One datum can be broadcast to all processors simultaneously, or a vector of $N$ items can be shifted cyclically among the processors. The technique that Stone used was for solving a system of equations is called "recursive doubling".

Chen and Kuck [1975] implemented an algorithm on an SIMD type machine to solve any linear system of the form $\underline{x}=\underline{c}+A \underline{x}$, where $A$ is an ( $n \times n$ ) strictly lower triangular matrix and $c$ is a constant column vector. However, as the algorithm stands, all processors execute the same operation at the same time. They show that $O\left(\log _{2}^{2} n\right)$ time steps are required using $O\left(n^{3}\right)$ processors.

Following on the work done by Stone, Clint and Perrot [1980] presented a solution of a wide variety of problems in numerical mathematics which needed the solution of sets of linear equations. Their algorithms are designed for an array processor like the ILLIAC IV,

SIMD type computers. In their implementation of parallel iterative methods like the Jacobi and Gauss-Seidel, each processor executes one component of the vector $x$. Hence, all the components can be evaluated simuitaneously at any one time.

In this thesis programs are run on an MIMD type computer whose processors act asynchronously, therefore iterative methods for solving a system of linear equations have to be constructed in asynchronous form, i.e., asynchronous iterative algorithms. In asynchronous iterative algorithms, each process in one iteration has to compute different and independent subsets of the components using their initial values stocedin the shared mempry. The values obtained in one iteration are used in the next iteration whilst computing the sawn components. Since the perforinance of the availuble processors are not tife same, each processorcan use at any computation time the values of the components that are evaluated and released from the previous iteration. On the other händ, when the value of the component is not available, the processior can then use the current value of that which was used in the previous iteration. As an example, Figure 4.8 , shows two processors working asynchronously on a system of four equations.

$$
\begin{aligned}
& \begin{array}{r}
P_{1} \\
x_{1}=\left(x_{1}^{0}, x_{2}^{0}, x_{3}^{0}, x_{4}^{0}\right) \\
x_{2}^{1}=\left(x_{1}^{0}, x_{2}^{0}, x_{3}^{0}, x_{4}^{0}\right) \\
x_{1}^{2}=\left(x_{1}^{1}, x_{2}^{1}, x_{3}^{1}, x_{4}^{1}\right) \\
x_{2}^{2}=\left(x_{1}^{1}, x_{2}^{1}, x_{3}^{1}, x_{4}^{1}\right) \\
x_{1}^{3}=\left(x_{1}^{2}, x_{2}^{2}, x_{3}^{2}, x_{4}^{2}\right) \\
x_{2}^{2}=\left(x_{1}^{2}, x_{2}^{2}, x_{3}^{2}, x_{4}^{2}\right)
\end{array} \\
& \left\{\begin{aligned}
& P_{2} \\
& x_{3}^{1}=\left(x_{1}^{0}, x_{2}^{0}, x_{3}^{0}, x_{4}^{0}\right) \\
& x_{4}^{1}=\left(x_{1}^{0}, x_{2}^{0}, x_{3}^{0}, x_{4}^{0}\right) \\
& x_{3}^{2}=\left(x_{1}^{0}, x_{2}^{0}, x_{3}^{1}, x_{4}^{1}\right) \\
& x_{4}^{2}=\left(x_{1}^{0}, x_{2}^{0}, x_{3}^{1}, x_{4}^{1}\right) \\
& x_{3}^{3}=\left(x_{1}^{1}, x_{2}^{1}, x_{3}^{2}, x_{4}^{2}\right) \\
& x_{3}^{4}=\left(x_{1}^{1}, x_{2}^{1}, x_{3}^{2}, x_{4}^{2}\right)
\end{aligned}\right.
\end{aligned}
$$

SIGURE 4.8: Two processors working asynchronously on 4 equations - means end of execution

To represent an asynchronous iterative method that solves a system of equations we suppose that $F$ is an operator from $\mathbb{R}^{n}$ into itself, we want to find a vector $x$ in $\mathbb{R}^{n}$ which satisfies the system of equations represented by,

$$
\begin{equation*}
x=F(x) \tag{4.9.1}
\end{equation*}
$$

Chazan and Miranker [1969] introduced the chaotic relaxation scheme, which is a class of iterative methods for solving equation (4.9.1), where $F$ is linear operator given by,

$$
\begin{equation*}
F(x)=A \underline{x}+\underline{b}, \tag{4.9.2}
\end{equation*}
$$

where $\underline{x}$ is an unknown vector, $A$ is an ( $n \times n$ ) matrix of coefficient and $\underline{b}$ is a constant $n$-vector. They showed that iterations defined by a chaotic relaxation scheme converge to (4.9.1) if and only if $\rho(|A|)<1$, where $\rho(|A|)$ is the spectral radius of the matrix $A$. The motivation of defining chaotic relaxation is to account for the parallel implementation of iterative methods on multiprocessor systems so as to reduce communication and synchronisation between the cooperating processes. This reduction is obtained by not forcing the processes to follow a predetermined sequence of computations, but simply to allow a process, when starting the evaluation of a new iterate, to choose dynamically not only the components to be evaluated but also the values of the previous iterates used in the evaluation. The restriction in the chaotic relaxation scheme is that there must exist a fixed positive integer $S$ such that, in carrying out the evaluation of the ith iterate, a process cannot make use of any value of the components of the $j$ th iterate, if j<i-S.

Baudet [1978] introduced a class of asynchronous iterative methods in which the chaotic relaxation is considered as a special case and the
above restriction can be avoided. The class of asynchronous iterative methods defined by Baudet is as follows:

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

$$
F:=\left[\begin{array}{c}
f_{1}(x) \\
\vdots \\
\vdots \\
f_{n}(x)
\end{array}\right] \quad x:=\left[\begin{array}{c}
x_{1} \\
\vdots \\
x_{n}
\end{array}\right]
$$

Let $J:=\left(J_{j}\right)_{j=1}^{\infty}$ be a sequence of non-empty subsets of $\{1,2, \ldots, n\}$
and let $S=\left[\begin{array}{c}s(j) \\ \vdots \\ s_{n}(j)\end{array}\right]_{j=1}^{\infty}$ be a sequence of elements in $N^{n}$ ( $\mathbb{N}^{n}$ the set of all non-negative integers). Thus, a sequence $x(j)=\left[\begin{array}{c}x_{1}(j) \\ \vdots \\ x_{n}(j)\end{array}\right] \in \mathbf{R}^{n}$, for $1 \leqslant j \leqslant \infty$ is called an asynchronous iterative sequence if the sequence $(x(j))_{j=1}^{\infty}$ is determined by a quadruple $(F, X(O), J, S)$ in the following way:
(1) F,J,S as defined above, $X(0)=\left[\begin{array}{c}X_{1}(0) \\ \vdots \\ X_{n}(0)\end{array}\right]$
(2) $x_{i}(J)=\left\{\begin{array}{ll}x_{i}(j-1) & \text { if } i \notin J_{j} \\ f_{i}\left(x_{1}\left(S_{1}(j)\right), \ldots, x_{n}\left(S_{n}(j)\right)\right), & \text { if } i \in J_{j}\end{array}\right\}$
(3) $i$ occurs infinitely often in the sets $J_{j}, j=1,2, \ldots, 1 \leqslant i \leqslant n$.
(4) $\forall i \in\{1,2, \ldots, n\}\left[S_{i}(j) \leqslant j-1, j=1,2, \ldots\right.$, and $\left.\underset{j \rightarrow \infty}{\lim _{i m}} S_{i}(j)+\infty\right]$.

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.

From the above conditions it is clear that an asynchronous iteration corresponding to the operator $F$ and starting with a given vector $x(0)$ and defined by $J$ and $S$, will be denoted by ( $F, X(0), J, S$ ). Condition (2)
of the definition means that, to evaluate $X_{i}$ in the jth iteration, the recent values obtained from ( $j-1$ ) th is used if they are released, otherwise, the previous values obtained from early iterations are to be used in the evaluation of a new iterate. Condition (3) of the definition guarantees that no component of the vector $x$ be abandoned during any iterate. While the first part of condition (4) states that only components of the previous iterate can be used in the evaluation of a new iterate, and the second part states that the most recent values are used instead of the values of an early iterate.

For example, the point Jacobi method defined in the operator $F$ with the initial approximation $\mathrm{x}(0)$ can be represented by the asynchronous iteration ( $F, X(O), J, S$ ), where

$$
\begin{aligned}
& J_{i}=\{1, \ldots, n\}, \text { for } j=1,2, \ldots, \\
& S_{i}(j)=j-1, \quad \text { for } j=1,2, \ldots, \text { and } i=1,2, \ldots, n .
\end{aligned}
$$

The same point Jacobi method can be equivalently represented by the asynchronous iteration where,

$$
\begin{aligned}
& J_{j}=\{1+(j-1 \bmod n)\}, \text { for } j=1,2, \ldots, \\
& S_{i}(j)=n\lfloor(j-1) / n\rfloor, \text { for } j=1,2, \ldots \text { and } i=1,2, \ldots, n .
\end{aligned}
$$

In the first case, this means that all the components $x_{1}, x_{2}, \ldots, x_{n}$ are evaluated at once and this, presumably, will be done by one computational process with the fact that the components of the new iterate, say $i$, cannot be evaluated until the values from the iterate ( $j-1$ ) have been obtained. While in the asynchronous Jacobi (second case), 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 value of these components at that time is
considered for that computation.
To ensure the convergence of the asynchronous iterative method, Baudet [1978] shows that, if $F$ is a contracting operator on a closed subset $D$ of $\mathbb{R}^{n}$ and if $F(D) \subset D$, then any asynchronous iteration $(F$, $X(0), J, S)$ corresponding to $F$ and starting with vector $X(O)$ in $D$ converges to the unique fixed point of $F$ in $D$.

Experimental results for solving linear systems iteratively using asynchronous iterative methods (such as Jacobi, Gauss-Seidel and S.O.R.) on an MIMD parallel computer can be found in Baudet [1978], Barlow and Evans [1982] and Yousif [1983].

### 4.10 THE FOUR-POINT EXPLICIT BLOCK ITERATIVE METHOD

Implicit block iterative methods were introduced in Section 4.8; in this section we present another approach of using a small group of points of fixed size, i.e., where groups of a certain number of individual equations (mesh point) and treated explicitly and similar to the way a single point is treated in the point iterative method. Evans and Biggins [1982] developed the four-point block iterative scheme and applied it 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, the groups themselves being ordered in red-black ordering as shown in Figure 4.9.


FIGURE 4.9

Because the blocks are taken in red-black ordering, the matrix has block Property (A) and is also block consistently ordered. Besides if the blocks are taken in natural ordering, the matrix is also block consistently ordered. It follows that the full theory of block S.O.R. method applies for both orderings.

Block iterative methods are also known as implicit methods since the solution of a whole group of points can be found at a time, as opposed to the point iterative methods where only one point is considered. Normally, implicit methods have larger convergence rates than those of explicit methods, at the cost of some extra computation involving the blocks in each iteration.

To derive the explicit block S.O.R. equations for the model problem, we consider the 4 mesh points as shown in Figure 4.10 .


## FIGURE 4.10

where $k=(\ell N+1)(2)(\ell+1) N-1$ and $\ell=O(2) N-2$. Also, we have that $N$ is an even number and $G_{M}$ of the definition (4.8.1), $M=1,2, \ldots, \frac{N^{2}}{4}$, such that each $G_{M}$ consists of the four elements, $\{k, k+1, N+k, N+k+1\}$ though, the
matrix $A_{M, M}$ is of order 4 and of the form,

$$
A_{M, M}=\left[\begin{array}{rrrr}
4 & -1 & -1 & 0  \tag{4.10.1}\\
-1 & 4 & 0 & -1 \\
-1 & 0 & 4 & -1 \\
0 & -1 & -1 & 4
\end{array}\right], M=1,2, \ldots, N^{2} / 4
$$

Thus, for the model problem, by using the mesh points shown in Figure 4.10, and from equation (4.8.8) the explicit group Jacobi method can be written as,

$$
\begin{align*}
& x_{k}^{(n+1)}=\frac{1}{24}\left[7\left(x_{k-N}^{(n)}+x_{k-1}^{(n)}\right)+2\left(x_{k-N+1}^{(n)}+x_{k+2}^{(n)}+x_{\left.\left.2 N+k^{(n)}+x_{N+k-1}^{(n)}\right)+x_{N+k+2}^{(n)}+x_{2 N+k+1}^{(n)}\right],}^{x_{k+1}^{(n+1)}=\frac{1}{24}\left[7\left(x_{k-N+1}^{(n)}+x_{k+2}^{(n)}\right)+2\left(x_{k-N}^{(n)}+x_{k-1}^{(n)}+x_{N+k+2}^{(n)}+x_{2 N+k+1}^{(n)}\right)+\left(x_{2 N+k}^{(n)}+x_{N+k-1}^{(n)}\right)\right],}\right.\right. \\
& x_{N+k}^{(n+1)}=\frac{1}{24}\left[7\left(x_{2 N+k}^{(n)}+x_{N+k-1}^{(n)}\right)+2\left(x_{k-N}^{(n)}+x_{k-1}^{(n)}+x_{N+k+2}^{(n)}+x_{2 N+k+1}^{(n)}\right)+x_{k-N+1}^{(n)}+x_{k+2}^{(n)}\right], \\
& x_{N+k+1}^{(n+1)}=\frac{1}{24}\left[7\left(x_{N+k+2}^{(n)}+x_{2 N+k+1}^{(n)}\right)+2\left(x_{k-N+1}^{(n)}+x_{k+2}^{(n)}+x_{2 N+k}^{(n)}+x_{N+k-1}^{(n)}\right)+\right.
\end{align*}
$$

$$
\left.x_{k-N}^{(n)}+x_{k-1}^{(n)}\right]
$$

where $k=(\ell N+1)(2)(\ell+1) N-1$ and $\ell=0(2) N-2$.
From equation (4.10.2) it can be seen that for the group of 4 points the Gauss-Seidel iterative method involves $\frac{N^{2}}{4}$ systems of equations of the form:

$$
\left[\begin{array}{rrrr}
4 & -1 & -1 & 0  \tag{4.10.3}\\
-1 & 4 & 0 & -1 \\
-1 & 0 & 4 & -1 \\
0 & -1 & -1 & 4
\end{array}\right]\left[\begin{array}{l}
x_{i, j} \\
x_{i, j+1} \\
x_{i+1, j} \\
x_{i+1, j+1}
\end{array}\right]^{(n+1)}\left[\begin{array}{l}
r_{i, j} \\
r_{i, j+1} \\
r_{i+1, j} \\
r_{i+1, j+1}
\end{array}\right]^{(n)} \text {, for } i, j=1(2) N,
$$

where, $\quad r_{i, j}^{(n)}=x_{i-1, j}^{(n+1)}+x_{i, j-1}^{(n+1)}$,

$$
r_{i, j+1}^{(n)}=x_{i-1, j+1}^{(n+1)}+x_{i+2, j}^{(n)}
$$

$$
\begin{equation*}
r_{i+1, j}^{(n)}=x_{i+1, j-1}^{(n+1)}+x_{i+2, j}^{(n)}, \tag{4.10.4}
\end{equation*}
$$

and

$$
r_{i+1, j+1}^{(n)}=x_{i+1, j+2}^{(n)}+x_{i+2, j+1}^{(n)}
$$

Now let,

$$
\begin{align*}
& s_{1}=r_{i, j}^{(n)}+r_{i, j}^{(n)}+r_{i+1, j+1}^{(n)}+r_{i+1, j+1}^{(n)},  \tag{4.10.5}\\
& s_{2}=r_{i, j+1}^{(n)}+r_{i, j+1}^{(n)}+r_{i+1, j}^{(n)}+r_{i+1, j}^{(n)} . \tag{4.10.6}
\end{align*}
$$

Hence, the solution of the system (4.10.3) can be found from the formula,

$$
\begin{align*}
& x_{i, j}^{(n+1)}=\frac{1}{24}\left(7 r_{i, j}^{(n)}+s_{2}+r_{i+1, j+1}^{(n)}\right), \\
& x_{i, j+1}^{(n+1)}=\frac{1}{24}\left(7 r_{i, j+1}^{(n)}+s_{1}+r_{i+1, j}^{(n)}\right),  \tag{4.10.7}\\
& x_{i+1, j}^{(n+1)}=\frac{1}{24}\left(7 r_{i+1, j}^{(n)}+s_{1}+r_{i, j+1}^{(n)}\right), \\
& x_{i+1, j+1}^{(n+1)}=\frac{1}{24}\left(7 r_{i+1, j+1}^{(n)}+s_{2}+r_{i, j}^{(n)}\right) .
\end{align*}
$$

When the over-relaxation factor $\omega$ is added, the application of the over-relaxation technique leads to the $(n+1)$ th iterate of the group of 4 points being redefined to give the S.O.R. formula as:

$$
\begin{aligned}
& x_{i, j}^{(n+1)}=x_{i, j}^{(n)}+\omega\left(x_{i, j}^{*}-x_{i, j}^{(n)}\right), \\
& x_{i, j+1}^{(n+1)}=x_{i, j+1}^{(n)}+\omega\left(x_{i, j+1}^{*}-x_{i, j+1}^{(n)}\right), \quad \text { for } i, j=1(2) n \\
& x_{i+1, j}^{(n+1)}=x_{i+1, j}^{(n)}+\omega\left(x_{i+1, j}^{*}-x_{i+1, j}^{(n)}\right), \\
& x_{i+1, j+1}^{(n+1)}=x_{i+1, j+1}^{(n)}+\omega\left(x_{i+1, j+1}^{*}-x_{i+1, j+1}^{(n)}\right),
\end{aligned}
$$

where $x^{*}$ represents the Gauss-Seidel solution $x^{(n+1)}$ defined in equation (4.10.7).

To calculate the amount of work required using this method,

[^6]$$
3 N^{2} \text { multiplications }+\frac{13}{2} \mathrm{~N}^{2} \text { additions }
$$
for $\mathrm{N}^{2}$ internal mesh points per iteration and assuming that the constant $\frac{1}{24}$ is stored beforehand. Equation (4.10.9) has been improved by solving $X_{i, j}^{(n+1)}$ and $x_{i+1, j+1}^{(n+1)}$, and to use these values to determine $X_{i, j+1}^{(n+1)}$ and $X_{i+1, j}^{(n+1)}$. Therefore, the average work per iteration for $\mathrm{N}^{2}$ internal mesh points including the over-relaxation process is,
$$
\frac{5 N^{2}}{2} \text { multiplications }+\frac{11 N^{2}}{2} \text { additions }
$$

### 4.11 THE 9-POINT EXPLICIT BLOCK ITERATIVE METHOD

In this method another grouping of the mesh points is suggested by considering each block to be formed from a group of 9 points as shown in Figure 4.ll. For this scheme of grouping, $N$ must be divisible by 3 . In this method, each subset $G_{M}, M=1,2, \ldots, \frac{N^{2}}{9}$, of Definition (4.8.1) consists of 9 elements, where $N^{2}$ represents the number of internal mesh points.


FIGURE 4.11

Suppose that the system of equations to be solved is derived from the two-dimensional Dirichlet problem (the model problem, see section 4.7), where the 5-point finite difference scheme shown in Figure (4.12) is used and given by the form,

$$
\begin{equation*}
\mathrm{Ax}=\underline{b} . \tag{4.11.1}
\end{equation*}
$$

The left hand side of the finite difference equation of such a system has the form,

$$
\begin{equation*}
x_{i, j}^{+\alpha_{1}} x_{i-1, j}+\alpha_{2} x_{i, j+1}^{+\alpha_{3} x_{i+1, j}+\alpha_{4} x_{i, j-1}} \tag{4.11.2}
\end{equation*}
$$

$$
\alpha_{3}
$$

FIGURE 4.12

For the mesh points shown in Figure 4.11 , the resulting block structure of the coefficient matrix $A$ of equation (4.11.1) is shown below,
where,

$$
A=\left[\begin{array}{ll:lc}
R_{0} & 0 & R_{2} & R_{3}  \tag{4.11.3}\\
0 & R_{0} & R_{1} & R_{4} \\
\hdashline R_{4} & R_{3} & R_{0} & 0 \\
R_{1} & R_{2} & 0 & R_{0}
\end{array}\right],
$$




From Figure 4.11, because the blocks are taken in red-black
ordering, the coefficient matrix A has block Property (A) and is also block consistently ordered. Besides, if the blocks are taken in natural ordering, the coefficient matrix is also block consistently ordered and has block Property (A). Again, it follows that the full theory of block S.O.R., method applies for both orderings.

Now, to derive the explicit block S.O.R. equations for the model problem, we calculate the transformed matrix $A$.

$$
\begin{equation*}
A^{E}=\left[\operatorname{diag}\left\{R_{0}\right\}\right]^{-1} A \tag{4.11.5}
\end{equation*}
$$

The matrix $\left[\operatorname{diag}\left\{R_{0}\right\}^{-1}\right.$ is simply $\operatorname{diag}\left\{R_{0}^{-1}\right\}$, and the inverse of matrix $R_{0}$ is given by,
where,

$$
\begin{aligned}
& d=4 \alpha_{1} \alpha_{2} \alpha_{3} \alpha_{4}\left[4\left(\alpha_{1} \alpha_{3}-\alpha_{2} \alpha_{4}\right)^{2}-2\left(\alpha_{1} \alpha_{3}+\alpha_{2} \alpha_{4}\right)+3\right]-2 \alpha_{1} \alpha_{3}\left(4 \alpha_{1}^{2} \alpha_{3}^{2}-6 \alpha_{1} \alpha_{3}+3\right) \\
& -2 \alpha_{2} \alpha_{4}\left(4 \alpha_{2}^{2} \alpha_{4}^{2}-6 \alpha_{2} \alpha_{4}+3\right)+1 . \\
& \beta_{1}=2 \alpha_{1} \alpha_{2} \alpha_{3} \alpha_{4}\left(4 \alpha_{1} \alpha_{3}-6 \alpha_{2} \alpha_{4}-1\right)-4 \alpha_{1} \alpha_{3}\left(\alpha_{1} \alpha_{3}-1\right)+\alpha_{2} \alpha_{4}\left(4 \alpha_{2}^{2} \alpha_{4}^{2}-4 \alpha_{2} \alpha_{4}+3\right)-1, \\
& \beta_{2}=2 \alpha_{1} \alpha_{2} \alpha_{3} \alpha_{4}\left(4 \alpha_{2} \alpha_{4}-2 \alpha_{1} \alpha_{3}-3\right)+4 \alpha_{1} \alpha_{3}\left(\alpha_{1} \alpha_{3}-1\right)-2 \alpha_{2}^{2} \alpha_{4}^{2}\left(2 \alpha_{2} \alpha_{4}-3\right)+1, \\
& \beta_{3}=2 \alpha_{1} \alpha_{2} \alpha_{3} \alpha_{4}\left(4 \alpha_{2} \alpha_{4}-6 \alpha_{1} \alpha_{3}-1\right)-4 \alpha_{2} \alpha_{4}\left(\alpha_{2} \alpha_{4}-1\right)+\alpha_{1} \alpha_{3}\left(4 \alpha_{1}^{2} \alpha_{3}^{2}-4 \alpha_{1} \alpha_{3}+3\right)-1, \\
& \beta_{4}=2\left[\left(2 \alpha_{1} \alpha_{3}-1\right)\left(2 \alpha_{2} \alpha_{4}-1\right)\right], \\
& \beta_{5}=4 \alpha_{1} \alpha_{3}\left(\alpha_{2} \alpha_{4}-\alpha_{1} \alpha_{3}+2\right)-2 \alpha_{2} \alpha_{4}-3, \\
& \beta_{6}=2 \alpha_{1} \alpha_{2} \alpha_{3} \alpha_{4}\left(4 \alpha_{1} \alpha_{3}-2 \alpha_{2} \alpha_{4}-3\right)+4 \alpha_{2} \alpha_{4}\left(\alpha_{2} \alpha_{4}-1\right)-2 \alpha_{2}^{2} \alpha_{4}^{2}\left(2 \alpha_{1} \alpha_{3}-1\right)+1, \\
& \beta_{7}=4 \alpha_{2} \alpha_{4}\left(\alpha_{1} \alpha_{3}-\alpha_{2} \alpha_{4}+2\right)-2 \alpha_{1} \alpha_{3}-3, \\
& \beta_{8}=2\left[\alpha_{1} \alpha_{3}\left(2 \alpha_{1} \alpha_{3}-4 \alpha_{2} \alpha_{4}-3\right)+\alpha_{2} \alpha_{4}\left(2 \alpha_{2} \alpha_{4}-3\right)+3\right] \text {, } \\
& \beta_{9}=-4 \alpha_{1} \alpha_{2} \alpha_{3} \alpha_{4}\left(2 \alpha_{1} \alpha_{3}-2 \alpha_{2} \alpha_{4}+1\right)+4 \alpha_{1}^{2} \alpha_{3}^{2}-4 \alpha_{2} \alpha_{4}\left(\alpha_{2} \alpha_{4}-1\right)-1, \\
& \beta_{10}=4 \alpha_{2} \alpha_{4}\left(\alpha_{2} \alpha_{4}-3 \alpha_{1} \alpha_{3}-1\right)+6 \alpha_{1} \alpha_{3}+1, \\
& \beta_{11}=4 \alpha_{1} \alpha_{2} \alpha_{3} \alpha_{4}\left(2 \alpha_{1} \alpha_{3}-2 \alpha_{2} \alpha_{4}-1\right)-4 \alpha_{1} \alpha_{3}\left(\alpha_{1} \alpha_{3}-1\right)+4 \alpha_{2}^{2} \alpha_{4}^{2}-1, \\
& \beta_{12}=4 \alpha_{1} \alpha_{3}\left(\alpha_{1} \alpha_{3}-3 \alpha_{2} \alpha_{4}-1\right)+6 \alpha_{2} \alpha_{4}+1, \\
& \gamma_{1}=2 \alpha_{1} \alpha_{2} \alpha_{3} \alpha_{4}\left[2\left(\alpha_{1} \alpha_{3}-\alpha_{2} \alpha_{4}\right)^{2}-\alpha_{1} \alpha_{3}-\alpha_{2} \alpha_{4}+3\right]-\alpha_{1} \alpha_{3}\left(4 \alpha_{1} \alpha_{3} \alpha_{3}-8 \alpha_{1} \alpha_{3}+5\right) \\
& -\alpha_{2} \alpha_{4}\left(4 \alpha_{2}^{2} \alpha_{4}^{2}-8 \alpha_{2} \alpha_{4}+5\right)+1, \\
& \gamma_{2}=-2 \alpha_{1} \alpha_{2} \alpha_{3} \alpha_{4}\left(4 \alpha_{1} \alpha_{3}-2 \alpha_{2} \alpha_{4}-3\right)+4 \alpha_{1} \alpha_{3}\left(\alpha_{1} \alpha_{3}-1\right)-\alpha_{2} \alpha_{4}\left(4 \alpha_{2}^{2} \alpha_{4}^{2}-8 \alpha_{2} \alpha_{4}+5\right)+1, \\
& \gamma_{3}=4 \alpha_{1}^{2} \alpha_{3}^{2}\left(\alpha_{2} \alpha_{4}-\alpha_{1} \alpha_{3}+2\right)-\alpha_{1} \alpha_{3}\left(8 \alpha_{2}^{2} \alpha_{4}^{2}-6 \alpha_{2} \alpha_{4}+5\right)+4 \alpha_{2} \alpha_{4}\left(\alpha_{2} \alpha_{4}-1\right)+1,
\end{aligned}
$$

and

$$
\gamma_{4}=4 \alpha_{1} \alpha_{2} \alpha_{3} \alpha_{4}\left(3-2 \alpha_{1} \alpha_{3}-2 \alpha_{2} \alpha_{4}\right)+4 \alpha_{1} \alpha_{3}\left(\alpha_{1} \alpha_{3}-1\right)+4 \alpha_{2} \alpha_{4}\left(\alpha_{2} \alpha_{4}-1\right)+1 .(4,11,8)
$$

The block structure of the matrix $A^{E}$ is the same as that of $A$ in equation (4.1l.3) with the submatrices $R_{0}$ replaced by the identity matrix $I$, and the submatrices $R_{i}, i=1,2,3,4$ replaced by $R_{0}^{-1} R_{i}$, which can be determined easily as follows:

$$
R_{O}^{-1} R_{1}=\frac{1}{d}\left[\begin{array}{cccccccc}
0 & 0 & \alpha_{1} \gamma_{1} & 0 & 0 & \alpha_{1} \alpha_{2} \beta_{3} & 0 & 0  \tag{4.11.9}\\
0 & \alpha_{1} \alpha_{2}^{2} \beta_{6} \\
0 & 0 & \alpha_{1}^{2} \beta_{1} & 0 & 0 & \alpha_{1} \alpha_{2} \beta_{4} & 0 & 0 \\
0 & 0 & \alpha_{1}^{3} \beta_{2} & 0 & 0 & \alpha_{1}^{3} \alpha_{2} \beta_{5} & 0 & 0 \\
0 & \alpha_{1} \alpha_{2}^{2} \beta_{1} \\
0 & 0 & \alpha_{1} \alpha_{4} \beta_{2} \beta_{2} & 0 & 0 & \alpha_{1} \gamma_{3} & 0 & 0 \\
0 & 0 & \alpha_{1}^{2} \alpha_{4} \beta_{4} & 0 & 0 & \alpha_{1}^{2} \beta_{11} & 0 & 0 \\
0 & \alpha_{1} \alpha_{2} \beta_{3} \alpha_{2} \beta_{4} \\
0 & 0 & \alpha_{1}^{3} \alpha_{4} \beta_{5} & 0 & 0 & \alpha_{1}^{3} \beta_{12} & 0 & 0 \\
0 & 0 & \alpha_{1} \alpha_{4}^{2} \beta_{6} & 0 & 0 & \alpha_{1} \alpha_{4} \beta_{3} & 0 & 0 \\
\alpha_{1}^{3} \alpha_{2} \beta_{5} \\
0 & 0 & \alpha_{1}^{2} \alpha_{4}^{2} \beta_{7} & 0 & 0 & \alpha_{1}^{2} \alpha_{4} \beta_{4} & 0 & 0 \\
0 & 0 & \alpha_{1}^{3} \alpha_{4}^{2} \beta_{8} & 0 & 0 & \alpha_{1}^{3} \alpha_{4} \beta_{5} & 0 & 0 \\
\alpha_{1} \beta_{1} \\
0 & \alpha_{1}^{3} \beta_{2}
\end{array}\right]
$$

$R_{0}^{-1} R_{2}=\frac{1}{\alpha}\left[\begin{array}{ccccccc}\alpha_{2}^{3} \beta_{6} & \alpha_{2}^{3} \alpha_{3} \beta_{7} & \alpha_{2}^{3} \alpha_{3}^{2} \beta_{8} & 0 & 0 & 0 & 0 \\ 0 & 0 \\ \alpha_{1} \alpha_{2}^{3} \beta_{7} & \alpha_{2}^{3} \beta_{10} & \alpha_{2}^{3} \alpha_{3} \beta_{7} & 0 & & & 0 \\ \alpha_{1} \alpha_{2}^{3} \beta_{8} & \alpha_{1} \alpha_{2}^{3} \beta_{7} & \alpha_{2}^{3} \beta_{6} & 0 & & & 0 \\ \alpha_{2}^{2} \beta_{3} & \alpha_{2} \alpha_{3} \beta_{4} & \alpha_{2}^{2} \alpha_{3}^{2} \beta_{5} & 0 & & 0 \\ \alpha_{1} \alpha_{2}^{2} \beta_{4} & \alpha_{2}^{2} \beta_{9} & \alpha_{2}^{2} \alpha_{3} \beta_{4} & 0 & 0 & 0 \\ \alpha_{1} \alpha_{2}^{2} \beta_{5} & \alpha_{1} \alpha_{2} \beta_{4} & \alpha_{2}^{2} \beta_{3} & 0 & & 0 \\ \alpha_{2} \gamma_{1} & \alpha_{2} \alpha_{3} \beta_{1} & \alpha_{2} \alpha_{3} \beta_{2} & 0 & & 0 \\ \alpha_{1} \alpha_{2} \beta_{1} & \alpha_{2} \gamma_{2} & \alpha_{2} \alpha_{3} \beta_{1} & 0 & & 0 \\ \alpha_{1}^{2} \alpha_{2} \beta_{2} & \alpha_{1} \alpha_{2} \beta_{1} & \alpha_{2} \gamma_{1} & 0 & 0 & 0 & 0 \\ 0\end{array}\right]$
and,

$$
R_{0}^{-1} R_{4}=\frac{1}{\alpha}\left[\begin{array}{lllllllll}
0 & 0 & 0 & 0 & 0 & 0 & \alpha_{4} \gamma_{1} & \alpha_{3} \alpha_{4} \beta_{1} & \alpha_{3}^{2} \alpha_{4} \beta_{2}  \tag{4.11.12}\\
0 & & & & 0 & \alpha_{1} \alpha_{4} \beta_{1} & \alpha_{4} \gamma_{2} & \alpha_{3} \alpha_{4} \beta_{1} \\
0 & & & & 0 & \alpha_{1}^{2} \alpha_{4} \beta_{2} & \alpha_{1} \alpha_{4} \beta_{1} & \alpha_{4} \gamma_{1} \\
0 & & & & 0 & \alpha_{4}^{2} \beta_{3} & \alpha_{3} \alpha_{4} \beta_{4} & \alpha_{3}^{2} \alpha_{4}^{2} \beta_{5} \\
0 & & 0 & 0 & \alpha_{1} \alpha_{4}^{2} \beta_{4} & \alpha_{4}^{2} \beta_{9} & \alpha_{3} \alpha_{4}^{2} \beta_{4} \\
0 & & & 0 & \alpha_{1}^{2} \alpha_{4}^{2} \beta_{5} & \alpha_{1} \alpha_{4}^{2} \beta_{4} & \alpha_{4}^{2} \beta_{3} \\
0 & & & & 0 & \alpha_{4}^{3} \beta_{6} & \alpha_{3} \alpha_{4}^{3} \beta_{7} & \alpha_{3}^{2} \alpha_{4}^{3} \beta_{8} \\
0 & & & & 0 & \alpha_{1} \alpha_{4}^{3} \beta_{7} & \alpha_{4}^{3} \beta_{10} & \alpha_{3} \alpha_{4}^{3} \beta_{7} \\
0 & 0 & 0 & 0 & 0 & 0 & \alpha_{1}^{2} \alpha_{4}^{3} \beta_{8} & \alpha_{1} \alpha_{4}^{3} \beta_{7} & \alpha_{4}^{3} \beta_{6}
\end{array}\right]
$$

It can be noticed that, if matrix $R_{i}$ has a column of zeros, so does the result matrix $R_{o}^{-1} R_{i}$, and where an element $\alpha_{i}$ occurs as the $(q, p)$ th element of $R_{i}$, the $q$ th column $R_{0}^{-1} R_{i}$ is the pth column of $R_{o}^{-1}$, multiplied by $\alpha_{i}$.

For the model problem the Dirichlet problem (4.7.8) we have,

$$
\alpha_{1}=\alpha_{2}=\alpha_{3}=\alpha_{4}=-\frac{1}{4},
$$

we have,
$R_{0}^{-1}=\frac{1}{56}\left[\begin{array}{ccccccccc}67 & 22 & 7 & 22 & 14 & 6 & 7 & 6 & 3 \\ 22 & 74 & 22 & 14 & 28 & 14 & 6 & 10 & 6 \\ 7 & 22 & 67 & 6 & 14 & 22 & 3 & 6 & 7 \\ 22 & 14 & 6 & 74 & 28 & 10 & 22 & 14 & 6 \\ 14 & 28 & 14 & 28 & 84 & 28 & 14 & 28 & 14 \\ 6 & 14 & 22 & 10 & 28 & 74 & 6 & 14 & 22 \\ 7 & 6 & 3 & 22 & 14 & 6 & 67 & 22 & 7 \\ 6 & 10 & 6 & 14 & 28 & 14 & 22 & 74 & 22 \\ 3 & 6 & 7 & 6 & 14 & 22 & 7 & 22 & 67\end{array}\right]$

Therefore,

|  |  | 0 | 0 | 67 | 0 | 0 | 22 | 0 | 0 | 7 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | 0 | 0 | 22 | 0 | 0 | 14 | 0 | 0 | 6 |
|  |  | 0 | 0 | 7 | 0 | 0 | 6 | 0 | 0 | 3 |
|  |  | 0 | 0 | 22 | 0 | 0 | 74 | 0 | 0 | 22 |
|  |  | 0 | 0 | 14 | 0 | 0 | 28 | 0 | 0 | 14 |
| $\begin{equation*} R_{0}^{-1} R_{1}=-\frac{1}{224} \tag{4.11.12} \end{equation*}$ |  | 0 | 0 | 6 | 0 | 0 | 10 | 0 | 0 | 6 |
|  |  | 0 | 0 | 7 | 0 | 0 | 22 | 0 | 0 | 67 |
|  | 0 | 0 | 0 | 6 | 0 | 0 | 14 | 0 | 0 | 22 |
|  | 0 | 0 | 0 | 3 | 0 | 0 | 6 | 0 | 0 | 7 |
|  | 7 | 7 | 6 | 3 |  |  |  |  |  |  |
|  | 6 | 61 | 10 | 6 |  |  |  |  |  |  |
|  | 3 | 3 | 6 | 7 |  |  |  |  |  |  |
| $\mathrm{R}_{0}^{-1} \mathrm{R}_{2}=-\frac{1}{224}$ |  | 221 | 14 | 6 |  |  | $\bigcirc$ |  |  |  |
|  |  | 142 | 28 | 14 |  |  |  |  |  |  |
|  |  |  | 14 | 22 |  |  |  |  |  |  |
|  |  | 67 | 22 | 7 |  |  |  |  |  |  |
|  |  | 227 | 74 | 22 |  |  |  |  |  |  |
|  |  |  | 22 | 67 |  |  |  |  |  |  |

(4.11.13)

Similarly, $\mathrm{R}_{\mathrm{O}}^{-1} \mathrm{R}_{3}$ and $\mathrm{R}_{\mathrm{O}}^{-1} \mathrm{R}_{4}$ can be obtained.
Explicit equations corresponding to the points that form the block of 9 points can also be derived by considering the area mesh as shown in Figure 4.13.


FIGURE 4.13

By applying the 5-point finite difference formula, the following system of equations is obtained:

$$
\begin{align*}
& 4 x_{1}=x_{2}+x_{4}+x_{A}+x_{B}, \\
& 4 x_{2}=x_{1}+x_{3}+x_{5}+x_{C}, \\
& 4 x_{3}=x_{2}+x_{6}+x_{D}+x_{E}, \\
& 4 x_{4}=x_{1}+x_{5}+x_{7}+x_{L},  \tag{4.11.14}\\
& 4 x_{5}=x_{2}+x_{4}+x_{6}+x_{8}, \\
& 4 x_{6}=x_{3}+x_{5}+x_{9}+x_{F}, \\
& 4 x_{7}=x_{4}+x_{8}+x_{J}+x_{\mathrm{K}}, \\
& 4 x_{8}=x_{5}+x_{7}+x_{9}+x_{I}, \\
& 4 x_{9}=x_{6}+x_{8}+x_{G}+x_{H},
\end{align*}
$$

and

Equation (4.11.14) can be rewritten as,

$$
\begin{align*}
& 4 x_{1}-x_{2} \quad-x_{4} \quad=x_{A}+x_{B} \text {, } \\
& -x_{1}+4 x_{2}-x_{3} \quad-x_{5} \quad=x_{c} \text {, } \\
& -x_{2}+4 x_{3} \quad-x_{6} \quad=x_{D}+x_{E}, \\
& \mathrm{~F}_{1} \quad+4 \mathrm{x}_{4}-\mathrm{x}_{5} \quad-\mathrm{x}_{7}  \tag{4.11.15}\\
& -x_{2} \quad-x_{4}+4 x_{5}-x_{6} \quad-x_{8} \quad=0 \\
& -x_{3} \quad-x_{5}+4 x_{6} \quad-x_{9} \quad=x_{F} . \\
& -x_{4} \quad+4 x_{7}-x_{8} \quad=x_{J}+x_{K} . \\
& -x_{5} \quad-x_{7}+4 x_{8}-x_{9} \quad=x_{I}, \\
& -x_{6} \quad-x_{8}+4 x_{9} \quad=X_{G}+H_{H} .
\end{align*}
$$

This can be written in matrix form as $A \underline{x}=\underline{b}$ and thus $\underline{x}=A^{-1} \underline{b}$ can then be given as,

$$
\begin{align*}
& x_{1}=\frac{1}{224}\left[67\left(X_{A}+X_{B}\right)+22\left(X_{C}+X_{L}\right)+7\left(X_{D}+X_{E}+X_{J}+X_{K}\right)+6\left(X_{F}+X_{I}\right)+3\left(X_{G}+X_{H}\right)\right], \\
& x_{2}=\frac{1}{112}\left[37 X_{C}+11\left(X_{B}+X_{A}+X_{D}+X_{E}\right)+7\left(X_{F}+X_{L}\right)+5 X_{I}+3\left(X_{G}+X_{H}+X_{J}+X_{K}\right)\right], \\
& x_{3}=\frac{1}{224}\left[67\left(X_{D}+X_{E}\right)+22\left(X_{C}+X_{F}\right)+7\left(X_{B}+X_{G}+X_{H}+X_{A}\right)+6\left(X_{I}+X_{L}\right)+3\left(X_{J}+X_{K}\right)\right], \\
& x_{4}=\frac{1}{112}\left[37 X_{L}+11\left(X_{A}+X_{B}+X_{J}+X_{K}\right)+7\left(X_{C}+X_{I}\right)+5 X_{F}+3\left(X_{D}+X_{E}+X_{G}+X_{H}\right)\right], \\
& x_{5}=\frac{1}{16}\left[2\left(X_{C}+X_{F}+X_{I}+X_{L}\right)+X_{B}+X_{A}+X_{D}+X_{E}+X_{G}+X_{H}+X_{J}+X_{K}\right], \\
& x_{6}=\frac{1}{112}\left[37 X_{F}+11\left(X_{D}+X_{E}+X_{G}+X_{H}\right)+7\left(X_{C}+X_{I}\right)+5 X_{L}+3\left(X_{A}+X_{B}+X_{J}+X_{K}\right)\right], \\
& x_{7}=\frac{1}{224}\left[67\left(X_{J}+X_{K}\right)+22\left(X_{I}+X_{L}\right)+7\left(X_{B}+X_{A}+X_{G}+X_{H}\right)+6\left(X_{C}+X_{F}\right)+3\left(X_{D}+X_{E}\right)\right], \\
& x_{8}=\frac{1}{112}\left[37 X_{I}+11\left(X_{G}+X_{H}+X_{J}+X_{K}\right)+7\left(X_{F}+X_{L}\right)+5 X_{C}+3\left(X_{A}+X_{B}+X_{D}+X_{E}\right)\right], \\
& x_{9}=\frac{1}{224}\left[67\left(X_{G}+X_{H}\right)+22\left(X_{F}+X_{I}\right)+7\left(X_{D}+X_{E}+X_{J}+X_{K}\right)+6\left(X_{C}+X_{L}\right)+3\left(X_{B}+X_{A}\right)\right] . \tag{4.11.16}
\end{align*}
$$

For the model problem, by using the mesh points as shown in Figure 4.14, the group Jacobi method can be written as,

$$
\begin{aligned}
& x_{k}^{(n+1)}=\frac{1}{224}\left[67\left(x_{k-N}^{(n)}+x_{(k-1)}^{(n)}\right)+22\left(x_{k-N+1}^{(n)}+x_{N+k-1}^{(n)}\right)+7\left(x_{k-N+2}^{(n)}+x_{k+3}^{(n)}+\right.\right. \\
& \left.x_{3 N+k^{(n)}}^{(n)}(n) \quad(n+k-1)+6\left(x_{N+k+3}^{(n)}+x_{3 N+k+1}^{(n)}\right)+3\left(x_{2 N+k+3}^{(n)}+x_{3 N+k+2}^{(n)}\right)\right], \\
& x_{k+1}^{(n+1)}=\frac{1}{112}\left[37 x_{k-N+1}^{(n)}+11\left(x_{k-N}^{(n)}+x_{k-1}^{(n)}+x_{k-N+2}^{(n)}+x_{k+3}^{(n)}\right)+7\left(x_{N+k+3}^{(n)}+x_{N+k-1}^{(n)}\right)\right. \\
& \left.+5 x_{3 N+k+1}^{(n)}+3\left(x_{2 N+k+3}^{(n)}+x_{3 N+k+2}^{(n)}+x_{3 N+k}^{(n)}+x_{2 N+k-1}^{(n)}\right)\right], \\
& x_{k+2}^{(n+1)}=\frac{1}{224}\left[67\left(x_{k-N+2}^{(n)}+x_{k+3}^{(n)}\right)+22\left(x_{k-N+1}^{(n)}+x_{N+k+3}^{(n)}\right)+7\left(x_{k-N}^{(n)}+x_{2 N+k+3}^{(n)}+\right.\right. \\
& \left.\left.x_{3 N+k+2}^{(n)}+x_{k-1}^{(n)}\right)+6\left(x_{3 N+k+1}^{(n)}+x_{N+k-1}^{(n)}\right)+3\left(x_{3 N+k}^{(n)}+x_{2 N+k-1}^{(n)}\right)\right], \\
& x_{N+k}^{(n+1)}=\frac{1}{112}\left[37 x_{N+k-1}^{(n)}+11\left(x_{k-N}^{(n)}+x_{\left.3 N+k^{(n)}+x_{2 N+k-1}^{(n)}+x_{k-1}^{(n)}\right)+7\left(x_{k-N+1}^{(n)}+x_{3 N+k+1}^{(n)}\right), ~(n)}^{(n)}\right.\right. \\
& \left.+5 x_{N+k+3}^{(n)}+3\left(x_{k-N+2}^{(n)}+x_{k+3}^{(n)}+x_{2 N+k+3}^{(n)}+x_{3 N+k+2}^{(n)}\right)\right] \text {, } \\
& x_{N+k+1}^{(n+1)}=\frac{1}{16}\left[2\left(x_{k-N+1}^{(n)}+x_{N+k+3}^{(n)}+x_{3 N+k+1}^{(n)}+x_{N+k-1}^{(n)}\right)+x_{k-N}^{(n)}+x_{k-1}^{(n)}+x_{k-N+2}^{(n)}+x_{k+3}^{(n)}\right. \\
& +x_{2 N+k+3}^{(n)}+X_{3 N+k+2}^{(n)}+X_{3 N+k}^{(n)}+x_{2 N+k-1}^{(n)}, \\
& x_{N+k+2}^{(n+1)}=\frac{1}{112}\left[37 x_{N+k+3}^{(n)}+11\left(x_{k-N+2}^{(n)}+x_{k+3}^{(n)}+x_{2 N+k+3}^{(n)}+x_{3 N+k+2}^{(n)}\right)+7\left(x_{k-N+1}^{(n)}+\right.\right. \\
& \left.\left.x_{3 N+k+1}^{(n)}\right)+5 x_{N+k-1}^{(n)}+3\left(x_{k-N}^{(n)}+x_{3 N+k}^{(n)}+x_{2 N+k-1}^{(n)}+x_{k-1}^{(n)}\right)\right] \text {, } \\
& x_{2 N+k}^{(n+1)}=\frac{1}{224}\left[6 7 \left(x_{3 N+k^{(n)}}^{\left(x_{2 N+k-1}^{(n)}\right)+22\left(x_{3 N+k+1}^{(n)}+x_{N+k-1}^{(n)}\right)+7\left(x_{k-N}^{(n)}+x_{k-1}^{(n)}+, ~+{ }_{2}^{(n)}\right.}\right.\right. \\
& \left.\left.x_{2 N+k+3}^{(n)}+x_{3 N+k+3}^{(n)}\right)+6\left(x_{k-N+1}^{(n)}+x_{N+k+3}^{(n)}\right)+3\left(x_{k-N+2}^{(n)}+x_{k+3}^{(n)}\right)\right], \\
& x_{2 N+k+1}^{(n+1)}=\frac{1}{112}\left[37 x_{3 N+k+1}^{(n)}+11\left(x_{2 N+k+3}^{(n)}+x_{3 N+k+2}^{(n)}+x_{3 N+k}^{(n)}+x_{2 N+k-1}^{(n)}\right)+7\left(x_{N+k+3}^{(n)}+\right.\right. \\
& \left.\left.+x_{N+k-1}^{(n)}\right)+5 x_{k-N+1}^{(n)}+3\left(x_{k-N}^{(n)}+x_{k-1}^{(n)}+x_{k-N+2}^{(n)}+x_{k+3}^{(n)}\right)\right],
\end{aligned}
$$

$$
\begin{align*}
x_{2 N+k+2}^{(n+1)}= & \frac{1}{224}\left[67\left(x_{2 N+k+3}^{(n)}+x_{3 N+k+2}^{(n)}\right)+22\left(x_{N+k+3}^{(n)}+x_{3 N+k+1}^{(n)}\right)+7\left(x_{k-N+2}^{(n)}+\right.\right. \\
& \left.\left.x_{k+3}^{(n)}+x_{3 N+k}^{(n)}+x_{2 N+k-1}^{(n)}\right)+6\left(x_{k-N+1}^{(n)}+x_{N+k-1}^{(n)}\right)+3\left(x_{k-N}^{(n)}+x_{k-1}^{(n)}\right)\right] \tag{4.11.17}
\end{align*}
$$

where $k=(\ell N+1)(3)(\ell+1) N-2$ and $\ell=0(3) N-3$.


## FIGURE 4.14

We now find the amount of computation that is needed to obtain the solution of the system of equations per iteration using the group of 9 points. From equation (4.11.17) it can be seen that the 9 -point Gauss-Seidel iterative method for the solution of our model problem involves $\frac{N^{2}}{9}$ systems of equations of the form,
where $\quad r_{i, j}^{(n)}=x_{i-1, j}^{(n+1)}+x_{i, j-1}^{(n+1)}$,

$$
\begin{aligned}
& r_{i, j+1}^{(n)}=x_{i-1, j+1}^{(n+1)}, \\
& r_{i, j+2}^{(n)}=x_{i-1, j+2}^{(n+1)}+x_{i, j+3}^{(n)}, \\
& r_{i+1, j}^{(n)}=x_{i+1, j-1}^{(n+1)}, \\
& r_{i+1, j+1}^{(n)}=0,
\end{aligned}
$$

$$
r_{i+1, j+2}^{(n)}=x_{i+1, j+3}^{(n)}
$$

$$
r_{i+2, j}^{(n)}=x_{i+2, j-1}^{(n+1)}+x_{i+3, j}^{(n)},
$$

$$
\text { and } \quad r_{i+2, j+1}^{(n)}=x_{i+3, j+1}^{(n)}
$$

$$
r_{i+2, j+2}^{(n)}=x_{i+3, j+2}^{(n)}+x_{i+2, j+3}^{(n)}
$$

Therefore, we have the Gauss-Seidel solution of equation (4.11.18), given by,

$$
\begin{aligned}
x_{i, j}^{(n+1)}= & \frac{1}{224}\left[67 r_{i, j}^{(n)}+22\left(r_{j, j+1}^{(n)}+r_{i+1, j}^{(n)}\right)+7\left(r_{i, j+2}^{(n)}+r_{i+2, j}^{(n)}\right)+6\left(r_{i+1, j+2}^{(n)}\right.\right. \\
& \left.\left.r_{i+2, j+1}^{(n)}\right)+3 r_{i+2, j+2}^{(n)}\right] \\
x_{i, j+1}^{(n+1)}= & \frac{1}{112}\left[37 r_{i, j+1}^{(n)}+11\left(r_{i, j}^{(n)}+r_{i, j+2}^{(n)}\right)+7\left(r_{i+1, j+2}^{(n)}+r_{i+1, j}^{(n)}\right)+5_{i+2, j+1}^{(n)}+3\left(r_{i+2, j}^{(n)}\right.\right.
\end{aligned}
$$

$$
\left.+r_{i+2, j+2}^{(n)}\right)
$$

$$
x_{i, j+2}^{(n+1)}=\frac{1}{224}\left[67 r_{i, j+2}^{(n)}+22\left(r_{i, j+1}^{(n)}+r_{i+1, j+2}^{(n)}\right)+7\left(r_{i, j}^{(n)}+r_{i+2, j+2}^{(n)}\right)+6\left(r_{i+2, j+1}^{(n)}\right.\right.
$$

$$
\left.+r_{i+1, j}^{(n)}\right)+3 r_{i+2, j}^{(n)}
$$

$$
x_{i+1, j}^{(n+1)}=\frac{1}{112}\left[37 r_{i+1, j}^{(n)}+11\left(r_{i, j}^{(n)}+r_{i+2, j}^{(n)}\right)+7\left(r_{i, j+1}^{\left.(n)+r_{i+2, j+1}^{(n)}\right)+5 r_{i+1, j+2}^{(n)}, ~(n)}\right.\right.
$$

$$
\left.+3\left(r_{i, j+2}^{(n)}+r_{i+2, j+2}^{(n)}\right)\right]
$$

$$
x_{i+1, j+1}^{(n+l)}=\frac{1}{16}\left[2\left(r_{i, j+1}^{(n)}+r_{i+1, j}^{(n)}+r_{i+1, j+2}^{(n)}+r_{i+2, j+1}^{(n)}\right)+r_{i, j}^{(n)}+r_{i, j+2}^{(n)}+r_{i+2, j}^{(n)}\right.
$$

$$
+x_{i+2, j+2}^{(n)}
$$

$$
x_{i+1, j+2}^{(n+1)}=\frac{1}{112}\left[37 r_{i+1, j+2}^{(n)}+11\left(r_{i, j+2}^{(n)}+r_{i+2, j+2}^{(n)}\right)+7\left(r_{i, j+1}^{(n)}+r_{i+2, j+1}^{(n)}\right)\right.
$$

$$
\left.+5 r_{i+1, j}^{(n)}+3\left(r_{i, j}^{(n)}+r_{i+2, j}^{(n)}\right)\right]
$$

$$
x_{i+2, j}^{(n+1)}=\frac{1}{224}\left[67 r_{i+2, j}^{(n)}+22\left(r_{i+2, j+1}^{(n)}+r_{i+1, j}^{(n)}\right)+7\left(r_{i, j}^{(n)}+r_{i+2, j+2}^{(n)}\right)+6\left(r_{i, j+1}^{(n)}\right.\right.
$$

$$
\left.\left.+r_{i+1, j+2}^{(n)}\right)+3 r_{i, j+2}^{(n)}\right]
$$

$$
x_{i+2, j+1}^{(n+1)}=\frac{1}{112}\left[37 r_{i+2, j+1}^{(n)}+11\left(r_{i+2, j+2}^{(n)}+r_{i+2, j}^{(n)}\right)+7\left(r_{i+1, j}^{(n)}+r_{i+1, j+2}^{(n)}\right)\right.
$$

$$
\left.+5 r_{i, j+1}^{(n)}+3\left(r_{i, j}^{(n)}+r_{i, j+2}^{(n)}\right)\right]
$$

$$
x_{i+2, j+2}^{(n+1)}=\frac{1}{224}\left[67 r_{i+2, j+2}^{(n)}+22\left(r_{i+1, j+2}^{(n)} r_{i+2, j+1}^{(n)}\right)+7\left(r_{i, j+2}^{(n)}+r_{i+2, j}^{(n)}\right)\right.
$$

$$
\begin{equation*}
\left.+6\left(r_{i, j+1}^{(n)}+r_{i+1, j}^{(n)}\right)+3 r_{i, j}^{(n)}\right] \tag{4.11.20}
\end{equation*}
$$

By the application of the over-relaxation technique on the $(n+1)$ th iteration of the group of 9 points we have the following formulae,

$$
\begin{aligned}
x_{i+l, j+k}^{(n+l)}=x_{i+\ell, j+k}^{(n)}+w\left(x_{i+\ell, j+k}^{*}-x_{i+\ell, j+k}^{(n)}\right) & \text { for } i, j=1(3) N \text { and } \\
& k, \ell=0(1) 2
\end{aligned}
$$

(4.11.21)
where $X^{*}$ represents the Gauss-Seidel solution $X^{(n+1)}$ defined by equation (4.11.20). Therefore this scheme requires a total of,

$$
\begin{equation*}
\frac{59 \mathrm{~N}^{2}}{9} \text { multiplications }+\frac{85}{9} \mathrm{~N}^{2} \text { additions. } \tag{4.11.22}
\end{equation*}
$$

We can improve the above computation work by, first calculating the 4 points $x_{i, j+1}^{(n+1)}, x_{i+1, j}^{(n+1)}, x_{i+1, j+2}^{(n+1)}$ and $x_{i+2, j+1}^{(n+1)}$ from equation (4.11.20) then apply the 5 point finite difference formula to the remaining five points. Hence we set,

$$
\begin{align*}
& s_{1}=x_{i, j+1}^{(n+1)}+x_{i+1, j}^{(n+1)}  \tag{4.11.23}\\
& s_{2}=x_{i, j+1}^{(n+1)}+x_{i+1, j+2}^{(n+1)}  \tag{4.11.24}\\
& s_{3}=x_{i+1, j}^{(n+1)}+x_{i+2, j+1}^{(n+1)} \tag{4.11.25}
\end{align*}
$$

and

$$
\begin{align*}
& s_{4}=x_{i+2, j+1}^{(n+1)}+x_{i+1, j+2}^{(n+1)}  \tag{4.11.26}\\
& x_{i, j}^{(n+1)}=\frac{1}{4}\left(r_{i, j}^{n}+s_{1}\right), \\
& x_{i, j+2}^{(n+1)}=\frac{1}{4}\left(r_{i, j+2}^{(n)}+s_{2}\right), \\
& x_{i+1, j+1}^{(n+1)}=\frac{1}{4}\left(s_{1}+s_{4}\right),  \tag{4.11.27}\\
& x_{i+2, j}^{(n+1)}=\frac{1}{4}\left(r_{i+2, j}^{(n)}+s_{3}\right), \\
& x_{i+2, j+2}^{(n+1)}=\frac{1}{4}\left(r_{i+2, j+2}^{(n)}+s_{4}\right),
\end{align*}
$$

so the amount of work per iteration for this method including the overrelaxation process is now,

$$
\begin{equation*}
\frac{38}{9} \mathrm{~N}^{2} \text { multiplications }+\frac{59}{9} \mathrm{~N}^{2} \text { additions. } \tag{4.11.28}
\end{equation*}
$$

### 4.12

## EXPERIMENTAL RESULTS OF THE BLOCK ITERATIVE METHODS

The parallel version of both the basic 9 -point block and the basic 4 -point block iterative methods have been implemented on the NEPTUNE system. These parallel versions are used to solve the model problem, i.e., the two-dimensional Dirichlet problem on a square grid of mesh points and the approximate solution is found by solving a linear system of equations, $A \underline{x}=\underline{b}$, where $A$ is $a(n \times n)$ sparse matrix and $\underline{b}$ is the vector obtained from the boundary conditions. To ensure that the solution to the model problem can be obtained, i.e. the solution converges then the spectral radius of matrix $A, \rho(A)$ should be less than 1 (see Section 4.2). The parallel versions of the 9-point block iterative methods were implemented using equations (4.11.19), (4.11.20) and (4.11.21) while the parallel versions of the 4 -point block iterative methods were implemented by using equations (4.10.4), (4.10.5), (4.10.6), (4.10.7) and (4.10.8). In general in all these versions, the model problem is solved by decomposing it into many subsets that are assigned to the different processors which can then be run in parallel. Obviously, different versions may give different results in the running time overheads, number of iterations needed for convergence and in the speedup ratios, which are studied and compared later. In the different parallel versions, two mesh sizes are evaluated, these sizes are $h^{-1}=25$ and $h^{-1}=37$. For the mesh point size $h^{-1}=25$ we need to evaluate a (24 $\times 24$ ) sparse matrix which is obtained by using the finite difference method (see Section (4.3), (4.10) and (4.11)). Similarly, for the mesh point size $h^{-1}=37$ we need to evaluate a $(36 \times 36)$ sparse matrix. Therefore, in general for mesh point size $h^{-1}=N+1$ we need to solve an $M=\left(N^{\times} N\right)$ matrix linear system in a multiprocessor of $P$ processors.

The natural way to evaluate these points is done by processing a fixed amount of work ( $M$ ́M) points by each process which is carried out by allocating $r=\frac{N}{p}$ lines (rows) of the matrix to each process. This implies that the first $r$ rows are assigned to process 1 , the second $r$ rows are assigned to process 2 and so on. In this approach, the rows are processed sequentially even though the components within each row are treated three at a time in the case of the 9 -point block iterative method or pairwise in the case of the 4 -point block iterative method. This method is more commonly known as sequential decomposition. In sequential decomposition, shared memory should be used to hold the input component values. These values can then be accessed by different processes. Each process iterates on its subset permanently, but it needs to read all its components before the start of the iteration. Then it releases all the values of the components for the next iteration. It is possible that a process cannot obtain the most recent value of its component, due to the time difference needed by each process to complete its work. This means that when a process $P$ is busy updating the values of its components the other processes cannot use the specific components when they are required until the update is completed. In this case, process $P$ will iterate using the old values which is related to that component. As a consequence to that situation an extra iteration will be needed to obtain the solution within a required accuracy. Sequential decomposition is considered as a good strategy, because when the mesh points subsets are allocated to different processes, all the related neighbouring points are computed sequentially within that subset, except for the rows on the boundary of the subset which have their
related rows in different subsets and which are carried out by other processes. Therefore, all the required new values of the related components are in the same subset. Similar results should be obtained for a larger number of mesh points, due to the capability of the parallel system being exploited when it is fully loaded.

In this section, two parallel versions have been programmed and implemented for both parallel 9-point block iterative methods and the parallel 4-point block iterative method using an synchronously and asynchronously approach with each parallel version. Also, two different $\omega$ values are used while running each algorithm, these values are $\omega=1.0$, i.e. similar to the Gauss-Seidel method, and for $\omega=\omega$ opt (optimal $\omega$ ), i.e. similar to the S.O.R. iterative method. Optimal $\omega$ is obtained from the experiments by choosing the one that gives the best running time. In all the parallel versions, the blocks within each subset are taken in both natural and red-black ordering. In our implementation the number of processes (parallel paths) are taken to be less or equal to the number of processors available and the accuracy value ( $\varepsilon$ ) taken to be equal to $10^{-5}$. The results shown in this section (such as timing, number of iterations, ...) are an average of many runs.

The Parallel 9-Point Block Iterative Method (Version One)
In this version of the 9-point block iterative method, the problem (coefficient matrix) is decomposed into subsets each of which are assigned to a parallel path. If $P$ is the number of available processors and $N$ is the size of the problem, i.e. the number of rows in the mesh which is divisible by $P$, then each path works on a subset of lines $N_{r}=\frac{N}{P}$. This means $P$ subsets are formed with each containing $N_{r}$ rows of
the original mesh points, where $N_{r}$ should be divisible by 3. Each subset will contain $b_{r}=\left\{\left(\frac{N_{r}}{3}\right)^{2}\right.$.P] blocks each with 9 points to be evaluated by each path. Each processor then computes its own subset by taking its blocks ( $b_{r}$ ) in the natural ordering, i.e., by taking up each successive three neighbouring rows at a time and all the blocks on these three lines are evaluated. When the blocks on the first three lines of the subset are evaluated, the second three lines are taken and all its blocks are also evaluated and the algorithm proceeds as before until all the lines in the subset are evaluated. Figure 4.15 shows an example when $N=12$ and $P=2$ and we have the number of lines in each subset $N_{r}=6$ and number of blocks in each subset $b_{r}=8$. Processors $P_{1}$ and $P_{2}$ evaluates their blocks in parallel and in natural ordering as shown in the figure below.


FIGURE 4.15

In this version, the nine points within each group are evaluated using equations (4.11.20) and (4.11.21), i.e. using the 13-point explicit formula. Version one has been implemented in both the asynchronous and synchronous approach. In the asynchronous approach, each processor run asynchronously on its subset without waiting for the other processors to complete their computations. In this case each processor iterates permanently on its subset until this and the other subsets which are carried out by other processors are converged. This approach has been implemented in Program 4.1, in this program the component values are maintained in a shared memory so that all the processors can obtain their subset value by accessing shared memory. A set of flags are also maintained in a shared memory, such that there is one flag for each processor which are used for convergence tests. At the end of each iteration, each processor checks to ensure that its components are obtained within the required accuracy, i.e. converged. If convergence is obtained, the processor sets its flag and tests the remaining flags to ensure that the other subsets that were run on different processors also converge. If any one of the other processors are not converged further iterations will be required, otherwise there is no need for further iteration. The results of the Parallel Version One algorithm are listed in Table 4.1 , where $1 \leqslant \omega \leqslant 1.9$, mesh size ( $24 \times 24$ ) and the number of paths as 1,2 and 4 . While Table 4.2 shows the results of this implementation by using mesh sizes ( $24 \times 24$ ) and ( $36 \times 36$ ) and for $\omega=1.0$ and $\omega=\omega_{\text {opt }}$ (optimal $\omega$ ) which is equal to $1.66,1.65$ for the $(24 \times 24)$ mesh size and 1.75 for the $(36 \times 36)$ mesh size.

Another strategy used to implement Version one is by evaluating the formed 9 -point blocks ( $b_{r}$ ) in red-black ordering instead of the
natural ordering. By red-black ordering we mean that odd blocks numbered $1,3,5 \ldots$ are evaluated first then the even numbered blocks 2,4,6... are evaluated next as shown in Figure 4.16. All assumptions that are applied to the natural ordering is also applied to the redblack ordering.


FIGURE 4.16

Figure 4.16 shows an example when $N=12$, number of processors $P=2$ and number of blocks in each subset $b_{r}=8$. Processors $P_{1}$ and $P_{2}$ evaluates their blocks simultaneously in red-black order as shown in Figure 4.16. The red-black strategy is programmed in Program 4.2 and the results of this program are listed in Table 4.3 where the mesh sizes $(24 \times 24)$ and $(36 \times 36)$ are used with $\omega=1.0$ and $\omega=\omega_{\text {opt }}$ (optimal $\omega$ ).

From the results in Tables 4.2 and 4.3 we observe that for both mesh sizes the running times for the asynchronous natural ordering (Program 4.1) are less than that of asynchronous red-black ordering (Program 4.2) and the speed-up ratios of the natural ordering is greater than that of the red-black ordering. Therefore, the natural ordering strategy appears to be better than that of the red-black ordering in the asynchronous implementation of the 9-point block using Version One. For that reason, we will choose the natural ordering among the two implementations for further investigation. We also noticed that for both strategies the speed-up ratios $\omega=\omega$ opt (optimal $\omega$ ) are higher than that for $\omega=1.0$. The timing results from Tables 4.2 and 4.3 using mesh size $(36 \times 36)$ with both $\omega=1.0$ and $\omega=\omega_{\text {opt }}$ are shown in Figure 4.17, while the speed-up results are shown in Figure 4.18.

| Mesh Size $(\mathrm{N} \times \mathrm{N})$ | $\varepsilon$ | No. of Processors | $\omega$ | $\begin{aligned} & \text { Time } \\ & \text { (seconds) } \end{aligned}$ | No. of iterations | Effective no. of parallel paths used by this processor |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $(24 \times 24)$ | $10^{-5}$ | 1 | $\begin{aligned} & 1.0 \\ & 1.1 \\ & 1.2 \\ & 1.3 \\ & 1.4 \\ & 1.5 \\ & 1.6 \\ & 1.7 \\ & 1.8 \\ & 1.9 \\ & \hline \end{aligned}$ | $\begin{array}{r} 1007.49 \\ 835.81 \\ 686.30 \\ 561.02 \\ 445.75 \\ 340.50 \\ 235.45 \\ 196.04 \\ 306.07 \\ 635.58 \\ \hline \end{array}$ | 201 167 137 112 89 68 47 39 61 127 | $\begin{aligned} & 1 \\ & 1 \\ & 1 \\ & 1 \\ & 1 \\ & 1 \\ & 1 \\ & 1 \\ & 1 \\ & 1 \end{aligned}$ |
|  |  | 2 | $\begin{aligned} & 1.0 \\ & 1.1 \\ & 1.2 \\ & 1.3 \\ & 1.4 \\ & 1.5 \\ & 1.6 \\ & 1.7 \\ & 1.8 \\ & 1.9 \end{aligned}$ | $\begin{array}{r} 509.28 \\ 420.85 \\ 348.52 \\ 282.48 \\ 225.60 \\ 171.60 \\ 117.76 \\ 96.48 \\ 157.90 \\ 396.06 \end{array}$ | $\begin{array}{r} 203 \\ 168 \\ 139 \\ 112 \\ 90 \\ 68 \\ 47 \\ 38 \\ 62 \\ 157 \end{array}$ | $\begin{aligned} & 1,1 \\ & 1,1 \\ & 1,1 \\ & 1,1 \\ & 1,1 \\ & 1,1 \\ & 1,1 \\ & 1,1 \\ & 1,1 \\ & 1,1 \end{aligned}$ |
|  |  | 4 | $\begin{aligned} & 1.0 \\ & 1.1 \\ & 1.2 \\ & 1.3 \\ & 1.4 \\ & 1.5 \\ & 1.6 \\ & 1.7 \\ & 1.8 \\ & 1.9 \end{aligned}$ | $\begin{array}{r} 255.65 \\ 213.70 \\ 176.00 \\ 145.80 \\ 113.78 \\ 91.31 \\ 70.62 \\ 48.39 \\ 89.28 \\ 257.56 \end{array}$ | 205 171 141 116 91 72 56 38 71 204 | $\begin{aligned} & 1,1,1,1 \\ & 1,1,1,1 \\ & 1,1,1,1 \\ & 1,1,1,1 \\ & 1,1,1,1 \\ & 1,1,1,1 \\ & 1,1,1,1 \\ & 1,1,1,1 \\ & 1,1,1,1 \\ & 1,1,1,1 \end{aligned}$ |

FIGURE 4.1: Results of the parallel 9-point block iterative method using natural ordering (Version One)

| Mesh Size $\left(\mathrm{N}^{\times} \mathrm{N}\right)$ | $\varepsilon$ | P | $\omega$ | (seconds) | No. of iterations | Speedup | Effective no. of parallel paths used by this processor |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| ( $24 \times 24$ ) | $10^{-5}$ | 1 2 4 | $\begin{aligned} & 1.0 \\ & 1.0 \\ & 1.0 \end{aligned}$ | $\begin{array}{r} 1007.49 \\ 509.28 \\ 255.65 \end{array}$ | $\begin{aligned} & 201 \\ & 203 \\ & 205 \end{aligned}$ | $\begin{gathered} 1 \\ 1.97826 \\ 3.94090 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1,1 \end{gathered}$ |
|  |  | 1 2 4 | $\begin{aligned} & 1.66 \\ & 1.66 \\ & 1.66 \end{aligned}$ | $\begin{array}{r} 160.10 \\ 80.39 \\ 40.10 \end{array}$ | $\begin{aligned} & 32 \\ & 32 \\ & 32 \end{aligned}$ | $\begin{gathered} 1 \\ 1.99154 \\ 3.99252 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1,1 \end{gathered}$ |
| $(36 \times 36)$ | $10^{-5}$ | 1 2 3 4 | $\begin{aligned} & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \end{aligned}$ | $\begin{aligned} & 4552.10 \\ & 2291.27 \\ & 1531.85 \\ & 1146.26 \end{aligned}$ | $\begin{aligned} & 403 \\ & 405 \\ & 406 \\ & 408 \end{aligned}$ | $\begin{gathered} 1 \\ 1.98672 \\ 2.97164 \\ 3.97126 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
|  |  | 1 2 3 4 | $\begin{aligned} & 1.75 \\ & 1.75 \\ & 1.75 \\ & 1.75 \end{aligned}$ | $\begin{aligned} & 524.50 \\ & 262.50 \\ & 175.22 \\ & 132.14 \end{aligned}$ | 47 <br> 46 <br> 46 <br> 47 | $\begin{gathered} 1 \\ 1.99771 \\ 2.99338 \\ 3.96928 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |

TABLE 4.2: The results of the asynchronous 9-point block iterative method obtained from Program 4.1, (Natural ordering, Version One)

| Mesh Size ( $\mathrm{N} \times \mathrm{N}$ ) | $\varepsilon$ | P | $\omega$ | $\begin{aligned} & \text { Time } \\ & \text { (seconds) } \end{aligned}$ | No. of iterations | $\begin{gathered} \text { Speed- } \\ \text { up } \end{gathered}$ | Effective no. of parallel paths used by this processor |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| (24×24) | $10^{-5}$ | $\begin{aligned} & 1 \\ & 2 \\ & 4 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.0 \\ & 1.0 \end{aligned}$ | $\begin{array}{r} 1021.58 \\ 517.06 \\ 260.94 \end{array}$ | $\begin{aligned} & 204 \\ & 205 \\ & 208 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.97575 \\ & 3.91500 \end{aligned}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1,1 \end{gathered}$ |
|  |  | $\begin{aligned} & 1 \\ & 2 \\ & 4 \end{aligned}$ | $\begin{aligned} & 1.65 \\ & 1.65 \\ & 1.65 \end{aligned}$ | $\begin{array}{r} 168.78 \\ 87.05 \\ 44.03 \end{array}$ | $\begin{aligned} & 34 \\ & 35 \\ & 35 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.93889 \\ & 3.83330 \end{aligned}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1,1 \end{gathered}$ |
| ( $36 \times 36$ ) | $10^{-5}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \end{aligned}$ | $\begin{aligned} & 4615.78 \\ & 2331.47 \\ & 1552.87 \\ & 1161.53 \end{aligned}$ | $\begin{aligned} & 409 \\ & 411 \\ & 412 \\ & 414 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.97947 \\ & 2.97196 \\ & 3.96985 \end{aligned}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
|  |  | 1 2 3 4 | $\begin{aligned} & 1.75 \\ & 1.75 \\ & 1.75 \\ & 1.76 \end{aligned}$ | $\begin{aligned} & 559.03 \\ & 281.64 \\ & 191.64 \\ & 146.95 \end{aligned}$ | $\begin{aligned} & 50 \\ & 51 \\ & 51 \\ & 52 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.98491 \\ & 2.91708 \\ & 3.80551 \end{aligned}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |

TABLE 4.3: The results of the asynchronous 9-point block iterative method obtained from Program 4.2, (Red-black, Version One)

Version One of the parallel 9-point block iterative method was implemented synchronously so that the blocks were taken in natural ordering. The natural ordering synchronous scheme was programmed in Program 4.3, where each processor evaluates (iterates) its own subset in the same manner as in asynchronous version with the exception that each processor synchronises itself after each iteration. In this case, each processor will wait for the other processors to finish their iteration and after all the processors are synchronised the convergent test is carried out by one processor (the master processor, processor 0
in the case of the NEPTUNE system). If the mesh components of previous iterations is not within the required accuracy a new iteration will be carried out by all the processors using their new subset values.

Otherwise the iterations are terminated which means convergence to the solution has been obtained.

Table 4.4 shows the results obtained from Program 4.3 using two mesh sizes $(24 \times 24)$ and $(36 \times 36)$ with $\omega$ taken as $\omega=1.0$ and $\omega=\omega_{\text {opt }}$ (optimal $\omega$ ), which equal to 1.66 in the case of the $(24 \times 24)$ mesh and 1.75 in the case of $(36 \times 36)$ mesh.

| $\begin{gathered} \text { Mesh Size } \\ (\mathrm{N} \times \mathrm{N}) \end{gathered}$ | $\varepsilon$ | P | $\omega$ | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | $\begin{aligned} & \text { Speed- } \\ & \text { up } \end{aligned}$ | Effective no.of parallel paths used by this processor |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| ( $24 \times 24$ ) | $10^{-5}$ | 1 2 4 | $\begin{aligned} & 1.0 \\ & 1.0 \\ & 1.0 \end{aligned}$ | $\begin{array}{r} 1009.19 \\ 510.93 \\ 256.34 \end{array}$ | $\begin{aligned} & 201 \\ & 200 \\ & 199 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.97520 \\ 3.93692 \end{gathered}$ | $\begin{gathered} 201 \\ 200,200 \\ 200,200,200,200 \end{gathered}$ |
|  |  | 1 2 4 | $\begin{aligned} & 1.66 \\ & 1.66 \\ & 1.66 \end{aligned}$ | $\begin{array}{r} 160.76 \\ 80.86 \\ 40.89 \end{array}$ | $\begin{aligned} & 32 \\ & 31 \\ & 30 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.98813 \\ & 3.93152 \end{aligned}$ | $\begin{gathered} 32 \\ 31,31 \\ 30,30,30,30 \end{gathered}$ |
| ( $36 \times 36$ ) | $10^{-5}$ | 1 2 3 4 | $\begin{aligned} & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \end{aligned}$ | $\begin{aligned} & 4553.17 \\ & 2292.82 \\ & 1533.09 \\ & 1148.04 \end{aligned}$ | $\begin{aligned} & 403 \\ & 402 \\ & 402 \\ & 400 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.98584 \\ 2.96341 \\ 3.95733 \end{gathered}$ | $\begin{gathered} 403 \\ 402,402 \\ 402,402,402 \\ 400,400,400,400 \end{gathered}$ |
|  |  | 1 2 3 4 | $\begin{aligned} & 1.75 \\ & 1.75 \\ & 1.75 \\ & 1.75 \end{aligned}$ | $\begin{aligned} & 530.42 \\ & 266.41 \\ & 177.51 \\ & 132.64 \end{aligned}$ | $\begin{aligned} & 47 \\ & 46 \\ & 46 \\ & 45 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.99099 \\ & 2.98811 \\ & 3.95733 \end{aligned}$ | 47 46,46 $46,46,46$ $45,45,45,45$ |

TABLE 4.4: The results of the synchronous 9-point block iterative method obtained from Program 4.3 (Natural ordering, Version One)

By comparing the results from Table 4.2 and 4.4 we notice that the time of Program 4.3 is greater than that of Program 4.1, i.e. evaluating
blocks in natural ordering asynchronously takes less time to converge than that of a synchronous evaluation and this is due to the synchronisation overheads needed after each iteration in the synchronous implementation. Also, it is clear that the speed-up ratios of asynchronous implementation is higher than that of a synchronous one. So we can say that, inspite of the efficient implementation of both the synchronous and asynchronous programs, the asynchronous natural order implementation gives better results in both time needed to converge and the speed-up ratios of the processors than that of synchronous implementation. This is due to the synchronisation overheads needed in the synchronous implementation. However, in the asynchronous implementation, all the processors obtain the most recent values of the components every time, because they are released as soon as they are updated. Figure 4.19, shows the timing results obtained from both Tables 4.2 and 4.4 , while Figure 4.20 shows the speed-up ratio results obtained from the same tables.

Paralle1 9-Point Block Iterative Method (Version Two)
In this version of the parallel 9-point iterative method we decompose the problem into subsets each of which are assigned to a parallel path, and where the number of parallel paths is equal to the number of co-operating processors. Also, the number of lines in each formed subset should be divisible by 3 and each processor then computes its own subset by taking up each successive three adjacent rows at a time and all the blocks on these three lines are evaluated. All the assumptions that were applied to Version One are also applied to Version Two with the exception of the way in which the components


The timing results of version one asynchronous 9-point block iterative method using natural and red-black ordering for mesh size $36 \times 36$


The speed-up results of version one asynchronous 9-point block iterative method using natural and red-black ordering for mesh size $36 \times 36$


The timing results of version one both asynchronous and synchronous 9-point block iterative method using natural ordering, mesh sizes $36 \times 36$ \& $24 \times 24$


FIGURE 4.20

Speed-up results of version one both asynchronous and synchronous 9-point block iterative method using natural ordering, mesh sizes $36 \times 36$ \& $24 \times 24$
within each block of the 9-point method are evaluated. In Version One of the parallel 9-point block iterative method each point within each block is evaluated using the 13 -point explicit formula as shown in equation (4.11.20), i.e. the points within each block are evaluated in the following order $(i, j),(i, j+1),(i, j+2),(i+1, j),(i+1, j+1)$, $(i+1, j+2),(i+2, j),(i+2, j+1)$ and $(i+2, j+2)$. While in version Two of the parallel 9 -point block iterative method, we start by evaluating the points $(i, j+1),(i+1, j),(i+1, j+2)$ and $(i+2, j+1)$ of each 9-point block using the 13 -point explicit formula (i.e., the same as that of Version One) and the remaining five points (i,j), (i,j+1), (i+1,j+1), $(i+2, j)$ and $(i+2, j+2)$ are evaluated using the 5-point finite difference formula as shown in equation (4.11.27). From Figure 4.21 we can see that the points labelled "1" are evaluated first using its outer boundary points (the points that are labelled with letters $A$ to L), i.e., each point of the 4 points labelled "I" will use the 12 boundary points, while evaluating its components as shown in equation (4.11.20). The remaining 5 points (labelled " 2 " in Figure 4.21) can now make use of the values obtained from the points labelled "l" using the 5-point finite difference formula (equation (4.11.27)), i.e. each point will use only 4 boundary points instead of 12 boundary points in the case of l3-point explicit formula (the first four points).

|  |  | $J$ | $I$ | $H$ |  |
| :--- | :--- | :--- | :--- | :--- | :--- |
| K |  | 2 | $I$ | 2 | $G$ |
| L |  |  |  |  | $F$ |
|  | 1 |  | 2 | $I$ | $E$ |
| A | 2 |  | 1 | 2 |  |
|  | B |  | C | D |  |

FIGURE 4.21

As in Version One, this version was implemented both asynchronously and synchronously. In the asynchronous implementation two strategies were used to evaluate the blocks within each subset, these strategies are the natural ordering and the red-black ordering (see Figure 4.15 and 4.16). Whilst in the synchronous implementation the natural ordering was implemented only. In these implementations each processor evaluates its subset and the convergence test was carried out on the NEPTUNE system as in that of Version One. For Version Two, the asynchronous strategy using both natural ordering and red-black ordering was implemented in Programs 4.4 and 4.5 respectively. The synchronous strategy with the natural ordering scheme for Version Two was implemented in Program 4.6. The experimental results of these programs using mesh size $(24 \times 24)$ are listed in Tables $4.5,4.6$ and 4.7 respectively.

By comparing the results from Table 4.5 and 4.6 we can see that the asynchronous natural ordering scheme takes less time than that of the asynchronous red-black ordering scheme and the speed-up factor of asynchronous natural ordering is higher than that of the red-black ordering. So we can say that in Version Two the asynchronous natural ordering is better than that of asynchronous red-black ordering. This is probably due to the overheads incurred by the system such as the interprocessor communication. Asynchronous natural ordering strategy was chosen among these two strategies and implemented synchronously in Program 4.6. Now comparing the results obtained from the asynchronous natural ordering (Table 4.5) and synchronous natural ordering (Table 4.7) we notice that the asynchronous strategy required less time than that of the synchronous one. While the speed-up ratios results for asynchronous strategy are greater than that of synchronous strategy
for both $\omega=1.0$ and $\omega \omega_{\text {opt }}$ (optimal $\omega$ ). Therefore we can say that in Version Two the asynchronous strategy also gives better results than that of synchronous one. This is due to the synchronisation overheads needed at the end of each iteration in the synchronous strategy and the usage of the recent values by the asynchronous strategy because they are relaxed as soon as they are updated.

| $\begin{aligned} & \text { Mesh Size } \\ & (N \times N) \end{aligned}$ | $\varepsilon$ | P | $\omega$ | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | Speedup | Effective no.of parallel paths used by this processor |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| (24×24) | $10^{-5}$ | $\begin{aligned} & 1 \\ & 2 \\ & 4 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.0 \\ & 1.0 \end{aligned}$ | $\begin{aligned} & 850.69 \\ & 431.32 \\ & 214.50 \end{aligned}$ | $\begin{aligned} & 201 \\ & 204 \\ & 205 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.98204 \\ 3.96592 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1,1 \end{gathered}$ |
|  |  | 1 2 4 | $\begin{aligned} & 1.42 \\ & 1.42 \\ & 1.42 \end{aligned}$ | $\begin{array}{r} 135.73 \\ 68.68 \\ 34.35 \end{array}$ | $\begin{aligned} & 32 \\ & 32 \\ & 32 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.97627 \\ 3.95138 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1,1 \end{gathered}$ |

TABLE 4.5: The results of the asynchronous 9-point block iterative method obtained from Program 4.4 (Natural ordering, Version Two)

| Mesh Size <br> $(\mathrm{N} \times \mathrm{N})$ | $\varepsilon$ | P | $\omega$ | Time <br> (seconds) | No.of <br> iter- <br> ations | Speed- <br> up | Effective no.of <br> paralle1 paths <br> used by this <br> processor |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $(24 \times 24)$ | $10^{-5}$ | 1 | 1.0 | 860.22 | 204 | 1.0 | 1 |
|  |  | 2 | 1.0 | 437.82 | 207 | 1.96478 | 1,1 |
|  |  | 4 | 1.0 | 219.77 | 208 | 3.91418 | $1,1,1,1$ |

TABLE 4.6: The results of the asynchronous 9-point block iterative method obtained from Program 4.5 (Red-black ordering, Version two)

| Mesh Size <br> $(\mathrm{N} \times \mathrm{N})$ | $\varepsilon$ | P | $\omega$ | Time <br> (seconds) | No.of <br> iter- <br> ations | Speed- <br> up | Effective no.of <br> parallel paths <br> used by this <br> processor |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $(24 \times 24)$ | $10^{-5}$ | 1 | 1.0 | 851.81 | 201 | 1.0 | 201 |
|  |  | 2 | 1.0 | 433.08 | 201 | 1.96687 | 201,201 |
|  |  | 4 | 1.0 | 215.53 | 198 | 3.95216 | $198,198,198,198$ |

TABLE 4.7: The results of the synchronous 9-point block iterative method obtained from Program 4.6 (Natural ordering, Version Two)

To compare both Version One and Two we take the best results of both versions and compare them. Both versions give the best results when using the asynchronous with natural ordering strategy. From the results in Table 4.2 and 4.5 when the mesh size is equal to ( $24 \times 24$ ) we can notice that the time required by Version Two is less than that of Version One. This is mainly from the way in which the components within each 9-point block are evaluated in both versions. In Version One each point is evaluated using the l3-point explicit formula, while in Version Two the first four points are evaluated using the 13 -point explicit formula and the remaining five points were evaluated using the 5 -point finite difference formula which means that less operations are required in Version Two than that of Version One. Also in Version Two, by using the most recent values of the first four points in evaluating the remaining five points within each 9-point block a greater rate of convergence is achieved since for these points a Gauss-Seidel approach is used. On the other hand, the speed-up ratios for Version One when $\omega=1.0$ are less than that of Version Two, while for $\omega=\omega_{\text {opt }}$ (optimal $\omega$ )
the speed-up ratios for Version One is greater than that of Version Two and optimal $\omega$ values in both versions are different. Figure 4.22 shows the timing results obtained from both Table 4.2 and 4.5 , while Figure 4.23 shows the speed-up ratio results obtained from the same tables.

To conclude from Version One and Two we can state that in all the implemented algorithms for the 9-point block iterative method the problem is decomposed into a subset each of which is assigned to a processor where the number of processors is always equal to the number of generated parallel paths. In all the implemented strategies (asynchronous and synchronous), each processor updates its components as soon as it is evaluated and it is made available to be used by the other processors by using a single shared array to hold the components' value. In the synchronous strategy the implementation is carried out by letting each processor wait at the end of each iteration for the other processors to finish their iteration in order to make sure that its new evaluated component values are available to the other processors to be used. This certainly 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. While the asynchronous strategy implemented is carried out in the same way as in the synchronous implementation but without waiting at the end of each iteration. Because of the way the implementation is carried out in both Version One and Two, we generally can say that in each Version where the timing results are concerned it does not matter whether the algorithm is synchronously or asynchronously implemented. Because in both Version One and Version Two we decompose the problem into almost
equal subsets and assign each one to different processors, this means that the amount of work carried out by each processor to evaluate any component is approximately the same, i.e. the complexity of evaluating any component by any processor is the same. To evaluate (update) a component in any block of the 9 points, its computational complexity (number of arithmetical operations) is equal to (13 Additions (A) + 8 Multiplications (M)), therefore for a mesh size equal to ( $N \times N$ ), there are $((13 A+8 M) N)$ operations per each line in each subset. Thus, for P parallel paths and $N_{r}=N / P$ lines in each subset, the total number of operations carried out by each processor is equal to $T=\left((13 A+8 M) \cdot N \cdot N_{r}\right)$. Besides the computational time $T$ there are extra overheads incurred by the system which degrades the algorithm performance in both the synchronous and asynchronous implementations. These overheads are the generation of parallel paths and the synchronisation at the end of each iteration cycle. These overheads may become significant, if for example we take the implementation of the synchronous and asynchronous methods by natural ordering (Version One) on the NEPTUNE system using four processor, $\omega=1.0$ and mesh size equal to $(36 \times 36)$. In the asynchronous algorithm, the problem converges after 403 iterations using a total of 4 parallel paths, while in the synchronous algorithm we reach the same answer after 403 iterations using a total of 808 parallel paths, since we need a synchronisation after each iteration. Thus, it is clear that the overheads may affect the performance of a parallel algorithm and specially the synchronisation overheads in this case. Hence for this reason we can say that the use of the asynchronous strategy is better suited for a MIMD computer.


FIGURE 4.22

The timing results of asynchronous natural ordering in both version one and two using results from table 4.1 \& 4.4 using mesh size $24 \times 24$


FIGURE 4.23

Speed-up results of asynchronous natural ordering in both version one and two using results from table 4.1 \& 4.4 using mesh size $24 \times 24$

Parallel 4-Point Block Iterative Method (Version One)
As in the parallel 9-point block iterative methods, in this version the problem is decomposed into subsets each of which are assigned to a parallel path which is run by a unique processor. The number of parallel paths will be equal to the number of co-operating processors. If $P$ is the number of the available processors and $N$ is the size of the problem (N divisible by P) then each parallel path evaluates in a subset of lines $N_{r}=N / P\left(N_{r}\right.$ should be divisible by 2). Each $P$ subset will contain $b_{r}=\left(\left(\frac{N_{r}}{2}\right)^{2}\right.$. P) blocks each with 4 points to be evaluated by a parallel path and run on a single processor. 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 are evaluated. When the blocks on these two lines are completed the next two lines in the subset are taken and the algorithm proceeds as before until all the lines in the subset are evaluated. In this version, the points within each block are evaluated using the 9-point finite difference explicit formula and implemented on the NEPTUNE system using equations (4.10.7) and (4.10.8) respectively. Version One of the parallel 4-point block iterative method has been implemented using different strategies as in Version One of the parallel 9-point block iterative method by taking the blocks within each subset in the natural ordering as well as in red-black ordering using both the synchronous and asynchronous approach. The principle of these strategies and the way in which they are performed and programmed are the same as in that of Version One of the parallel 9-point block iterative method.

Table 4.8 shows the results obtained from the implementation of Version One of the parallel 4-point block iterative method asynchronously
on the NEPTUNE system. The blocks within each subset are evaluated in natural ordering for both $\omega=1.0$ and $\omega=\omega$ opt (optimal $\omega$ ) using mesh sizes $(24 \times 24)$ and $(36 \times 36)$. Table 4.9 shows the results of the same strategy as that of Table 4.8 evaluating the blocks within each subset in red-black ordering instead of natural ordering. By comparing the results from Table 4.8 and 4.9 we notice that the running times taken by the problem to converge using asynchronous red-black ordering is higher than that of using the asynchronous natural ordering and the speed-up ratios of the natural ordering implementation is higher than the speed-up ratios of the red-black ordering implementation. Also, for both mesh sizes $(24 \times 24)$ and ( $36 \times 36$ ) the result shows that the speedup ratios of optimal $\omega$ ( $\omega_{\text {opt }}$ ) is higher than that of $\omega=1.0$ for both natural and red-black ordering. Thus we choose the natural ordering implementation as the best amongst the two implementations for the asynchronous strategy of Version One. Figure 4.24 shows the timing results from Tables 4.8 and 4.9 for the mesh size $(24 \times 24)$ with both $\omega=1.0$ and $\omega=\omega_{\text {opt }}$ while Figure 4.25 shows the speed-up ratios of both results.

| $\begin{gathered} \text { Mesh Size } \\ (N \times N) \end{gathered}$ | $\varepsilon$ | P | $\omega$ | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | $\begin{aligned} & \text { Speed- } \\ & \text { up } \end{aligned}$ | Effective no. of parallel paths used by this processor |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| (24×24) | $10^{-5}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \end{aligned}$ | $\begin{array}{r} 1075.76 \\ 541.38 \\ 363.15 \\ 271.09 \end{array}$ | $\begin{aligned} & 286 \\ & 288 \\ & 289 \\ & 289 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.98707 \\ 2.96230 \\ 3.96828 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
|  |  | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 1.7 \\ & 1.7 \\ & 1.71 \\ & 1.71 \end{aligned}$ | $\begin{array}{r} 150.36 \\ 75.40 \\ 50.40 \\ 37.63 \end{array}$ | $\begin{aligned} & 40 \\ & 39 \\ & 39 \\ & 39 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.99417 \\ 2.98333 \\ 3.99575 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| $(36 \times 36)$ | $10^{-5}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.0 \\ & 1.0 \end{aligned}$ | $\begin{aligned} & 4849.03 \\ & 2449.42 \\ & 1633.93 \end{aligned}$ | $\begin{aligned} & 572 \\ & 576 \\ & 577 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.97967 \\ 2.96771 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \end{gathered}$ |
|  |  | 1 2 3 | $\begin{aligned} & 1.79 \\ & 1.79 \\ & 1.79 \end{aligned}$ | $\begin{aligned} & 496.82 \\ & 248.56 \\ & 165.65 \end{aligned}$ | $\begin{aligned} & 59 \\ & 58 \\ & 56 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.99880 \\ 2.99922 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \end{gathered}$ |

TABLE 4.8: The results of the asynchronous 4-point block iterative method obtained from Version One using Natural ordering of blocks

| Mesh Size ( $\mathrm{N} \times \mathrm{N}$ ) | $\varepsilon$ | P | $\omega$ | $\begin{aligned} & \text { Time } \\ & \text { (seconds) } \end{aligned}$ | No. of iterations | Speedup | Effective no. of parallel paths used by this processor |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| (24×24) | $10^{-5}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \end{aligned}$ | $\begin{array}{r} 1091.24 \\ 552.08 \\ 368.88 \\ 276.35 \end{array}$ | $\begin{aligned} & 290 \\ & 293 \\ & 293 \\ & 294 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.97650 \\ 2.95825 \\ 3.94876 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
|  |  | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 1.71 \\ & 1.71 \\ & 1.71 \\ & 1.71 \end{aligned}$ | $\begin{array}{r} 152.99 \\ 80.79 \\ 54.19 \\ 40.62 \end{array}$ | $\begin{aligned} & 41 \\ & 43 \\ & 43 \\ & 43 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.89368 \\ 2.92322 \\ 3.76637 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| ( $36 \times 36$ ) | $10^{-5}$ | 1 2 3 | $\begin{aligned} & 1.0 \\ & 1.0 \\ & 1.0 \end{aligned}$ | $\begin{aligned} & 4924.03 \\ & 2487.52 \\ & 1660.75 \end{aligned}$ | $\begin{aligned} & 581 \\ & 585 \\ & 585 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.97949 \\ 2.96494 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \end{gathered}$ |
|  |  | 1 2 3 | $\begin{aligned} & 1.79 \\ & 1.79 \\ & 1.79 \end{aligned}$ | $\begin{aligned} & 497.56 \\ & 258.55 \\ & 173.62 \end{aligned}$ | $\begin{aligned} & 59 \\ & 61 \\ & 60 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.92440 \\ 2.86580 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \end{gathered}$ |

TABLE 4.9: The results of the asynchronous 4-point block iterative method obtained from version One using red-black ordering of blocks


The timing results of version one asynchronous 4-point block iterative method using natural \& red-black ordering for mesh size $24 \times 24$


Speed-up results of version one asynchronous 4-point block iterative method using natural and red-black ordering formesh size $24 \times 24$

However, Version One of the 4 -point block iterative method using the natural ordering strategy was implemented synchronously and its results are shown in Table 4.10. The natural ordering synchronous implementation was programmed in the same manner as that of Version One of the synchronous 9-point block iterative method, where each processor synchronises itself at the end of each iteration. Therefore, each processor will wait for the other processors to finish their iteration and the convergence test will be carried out by one processor (the master processor, processor 0 in the case of the NEPTUNE system). Another iteration will be carried out by all the processörs if convergence is not achieved. Table 4.10 shows the results of the synchronous parallel 4-point block iterative method obtained from Version One using the natural ordering using mesh sizes (24×24) and ( $36 \times 36$ ) for both $\omega=1.0$ and $\omega=o p t i m a l ~ w\left(\omega_{\text {opt }}\right)$.

By comparing the results from both Table 4.8 and 4.10 we notice that the time for the synchronous implementation is greater than that of the asynchronous implementation for both $\omega=1.0$ and $\omega=\omega_{\text {opt }}$. Also we notice that the speed-up ratios of the asynchronous implementation is higher than that of the synchronous implementation. This improvement in the asynchronous implementation is due to the synchronisation overheads at the end of each iteration in the synchronous implementation and the usage of the most recent component values all the time in the asynchronous strategy. Figures 4.26 and 4.27 show the timing and the speed-up ratios results obtained from both Table 4.8 and 4.10 respectively.

| Mesh size ( $\mathrm{N} \times \mathrm{N}$ ) | $\varepsilon$ | P | $\omega$ | $\left\{\begin{array}{c} \text { Time } \\ \text { (seconds) } \end{array}\right.$ | No. of iterations | $\begin{aligned} & \text { Speed- } \\ & \text { up } \end{aligned}$ | Effective no.of parallel paths used by this processor |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| (24×24) | $10^{-5}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \end{aligned}$ | $\begin{array}{r} 1075.95 \\ 545.48 \\ 366.53 \\ 274.94 \end{array}$ | $\begin{aligned} & 286 \\ & 285 \\ & 285 \\ & 284 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.97248 \\ & 2.93550 \\ & 3.91340 \end{aligned}$ | $\begin{gathered} 286 \\ 285,285 \\ 285,285,285 \\ 284,284,284,284 \end{gathered}$ |
|  |  | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 1.7 \\ & 1.7 \\ & 1.7 \\ & 1.7 \end{aligned}$ | $\begin{array}{r} 150.73 \\ 75.83 \\ 50.55 \\ 37.77 \end{array}$ | $\begin{aligned} & 40 \\ & 39 \\ & 39 \\ & 38 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.98774 \\ & 2.98180 \\ & 3.99073 \end{aligned}$ | $\begin{gathered} 40 \\ 39,39 \\ 39,39,39 \\ 38,38,38,38 \end{gathered}$ |
| $(36 \times 36)$ | $10^{-5}$ | 1 2 3 | $\begin{aligned} & 1.0 \\ & 1.0 \\ & 1.0 \end{aligned}$ | $\begin{aligned} & 4849.44 \\ & 2464.81 \\ & 1642.88 \end{aligned}$ | $\begin{aligned} & 572 \\ & 574 \\ & 571 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.96747 \\ & 2.95179 \end{aligned}$ | $\begin{gathered} 572 \\ 574,574 \\ 571,571,571 \end{gathered}$ |
|  |  | 1 2 3 | $\begin{aligned} & 1.79 \\ & 1.79 \\ & 1.79 \end{aligned}$ | $\begin{aligned} & 500.06 \\ & 251.30 \\ & 166.74 \end{aligned}$ | $\begin{aligned} & 59 \\ & 58 \\ & 56 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.98989 \\ 2.99904 \end{gathered}$ | $\begin{gathered} 59 \\ 58,58 \\ 56,56,56 \end{gathered}$ |

TABLE 4.10: The results of the synchronous parallel 4-point block iterative method obtained from Version One using Natural ordering


FIGURE 4.26

The timing results of version one asynchronous and synchronous 4-point block iterative method using Natural ordering for mesh size $24 \times 24$


FIGURE 4.27

Speed-up results of version one asynchronous and synchronous 4-point block iterative method using natural ordering for mesh size $24 \times 24$

## Parallel 4-Point Block Iterative Method (Version Two)

In Version Two of the parallel 4-point block iterative method we first evaluate two selected points from the 4 -point block then these two recent values are used to evaluate the remaining two points of the block. In Version One of the Parallel 4-point block iterative method the four points $(i, j),(i, j+1),(i+1, j)$ and $(i+1, j+1)$ are evaluated in the specified order using the 9-point explicit formula as shown in equations (4.10.7) and (4.10.8). While in Version Two of the parallel 4-point block iterative method we first evaluate the points (i,j) and ( $i+1, j+1$ ) using the 9 -point explicit formula (i.e. equations (4.10.7) and (4.10.8)) then use the most recent value of these two points to evaluate the remaining two points $(i+1, j)$ and $(i, j+1)$ using the 5-point finite difference formula as shown in Figure 4.28. From Figure 4.28, the points labelled "1" are evaluated first using its related outer boundary points (the points labelled with letters) using equations (4.10.7) and (4.10.8) The other two points labelled "2" are evaluated using its four known boundary points which include the two points labelled "l". This means that number of arithmetical operations required in Version Two are less than that of Version One.


FIGURE 4.28

As in Version One we decompose the problem into subsets each of which are assigned to a parallel path and run on a single processor by taking the number of parallel paths equal to the number of processors. Also Version Two was implemented using a synchronously and asynchronously strategy with natural and red-black ordering schemes for the asynchronous implementation and a natural ordering scheme in the synchronous implementation. The strategies and schemes were implemented in a similar way as in Version One but with a difference in the way in which the points within each block were evaluated. The results of Version Two asynchronous implementation on the NEPTUNE system using natural ordering with $\omega=1.0$ and $\omega=\omega_{\text {opt }}$ (optimal $\omega$ ) for mesh size (24×24) are displayed in Table 4.11. While Table 4.12 shows the results of the implementation of asynchronous red-black ordering using both $\omega=1.0$ and $\omega=\omega_{o p t}$ with mesh size (24×24).

From these two tables we can see that the asynchronous natural ordering results are better than that of the asynchronous red-black ordering. This is because the running time of asynchronous natural ordering is less than that of the asynchronous red-black ordering for both $\omega=1.0$ and $\omega=\omega_{\text {opt }}$. Also, the speed-up of the asynchronous natural ordering is higher than that of the asynchronous red-black ordering for both $\omega=1.0$ and $\omega=\omega_{\text {opt }}$. Thus, the asynchronous natural ordering will be chosen as best among these two strategies. Also, in both the asynchronous natural ordering and asynchronous red-black ordering better speed-up ratios are obtained when $\omega=\omega$ opt Version Two of the parallel 4-point block iterative method was also implemented synchronously. This implementation is the same as that of Version One, i.e. at the end of each iteration each processor will be synchronised with the other processors
to ensure the most recent component values are used by all the processors. Table 4.13 shows the results from computer runs using the synchronous natural ordering scheme using both $\omega=1.0$ and $\omega=\omega_{\text {opt }}$ for mesh size (24×24). By comparing the results from both the synchronous natural ordering, Table 4.13 and asynchronous natural ordering, Table 4.11 we notice that the running times of the asynchronous scheme is less than that of the synchronous scheme and the speed-up factors of the asynchronous scheme is higher than that of the synchronous scheme. This is due to the delay caused by the synchronisation times needed at the end of each iteration in the synchronous scheme and the usage of the most recent component values all the time in the asynchronous scheme. In this case, we can consider the asynchronous natural ordering is better than the synchronous natural ordering for the above reason.
$\left.\begin{array}{|c|c|c|c|c|c|c|c|}\hline \begin{array}{c}\text { Mesh Size } \\ (\mathrm{N} \times \mathrm{N})\end{array} & \varepsilon & \mathrm{P} & \omega & \begin{array}{c}\text { Time } \\ \text { (seconds) }\end{array} & \begin{array}{c}\text { No.of } \\ \text { iter- } \\ \text { ations }\end{array} & \begin{array}{c}\text { Speed- } \\ \text { up }\end{array} & \begin{array}{l}\text { Effective no.of } \\ \text { parallel paths } \\ \text { used by this }\end{array} \\ \hline(24 \times 24) & 10^{-5} & 1 & 1.0 & 987.99 & 286 & 1.0 & 1 \\ \text { processor }\end{array}\right]$

TABLE 4.11: The results of the asynchronous 4-point block iterative method obtained from Version Two using natural ordering

| Mesh Size <br> $(\mathrm{N} \times \mathrm{N})$ | $\varepsilon$ | P | Time <br> (seconds) | No.of <br> iter- <br> ations | Speed- <br> up | Effective no. <br> of parallel <br> paths used by <br> this processor |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $(24 \times 24)$ | $10^{-5}$ | 1 | 1.0 | 1003.17 | 291 | 1.0 | 1 |
|  |  | 2 | 1.0 | 505.66 | 293 | 1.98388 | 1,1 |
|  |  | 4 | 1.0 | 338.04 | 293 | 2.96761 | $1,1,1$ |
|  |  | 1.0 | 254.25 | 294 | 3.94561 | $1,1,1,1$ |  |

TABLE 4.12: The results of the asynchronous 4-point block iterative method obtained from version Two using red-black ordering

| Mesh Size <br> $(N \times N)$ | $\varepsilon$ | $\boldsymbol{P}$ | $\omega$ | Time <br> (seconds) | No.of <br> iter- <br> ations | Speed- <br> up | Effective no. <br> of parallel <br> paths used by <br> this processor |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $(24 \times 24)$ | $10^{-5}$ | 1 | 1.0 | 988.320 | 286 | 1.0 | 286 |
|  |  | 2 | 1.0 | 500.430 | 285 | 1.974949 | 285,285 |
|  |  | 4 | 1.0 | 335.34 | 284 | 2.94722 | $284,284,284$ |
| 4 | 1.0 | 252.18 | 283 | 3.91911 | $283,283,283,283$ |  |  |

TABLE 4.13: The results of the synchronous 4-point block iterative method obtained from Version Two using natural ordering

To compare both Version One and Two of the parallel 4-point block iterative method, we use the results obtained from the implementation of the natural asynchronous strategy in both versions because they give
the best results in each version. Now comparing the results from Tables (4.8) and (4.11) for mesh size (24×24) we notice that the time required by Version Two is less than that of Version One and this is due to the way in which the points in each 4 -point block are evaluated. In Version One, each point in the 4 -point block are evaluated using the 9-point explicit formula, while in version Two, the first two points in the 4 -point block are evaluated using the 9 -point explicit formula and the remaining two points are evaluated using the 5-point finite difference formula, which means, less computational operations are required in Version Two than that of Version One. Figure 4.29 shows that the timing results of both Version One and Two of parallel 4-point block iterative method using asynchronous natural ordering strategy using $\omega=1.0$ and $\omega=\omega$ opt with mesh size ( $24 \times 24$ ).

To conclude from Version One and Two of the parallel 4-point block iterative method, we generally can say that in each version where the timing results are concerned it does not matter whether the algorithm is synchronously or asynchronously implemented. Because in both versions we decompose the problem into almost equal subsets and assign each one to different processors, this means that the amount of work carried out by each processor to evaluate any component is approximately the same, i.e. the complexity of evaluating any component by any processor is the same.

In our implementations of the parallel 4-point block iterative methods we decompose the problem into equal subsets and each subset is assigned to a parallel path which runs on a single processor, this means that the same amount of work will be carried out by each processor, i.e. the complexity of evaluating any component by any processor is the


FIGURE 4.29

The timing results of asynchronous natural ordering in both version one and two of parallel 4 -point block iterative method for mesh size $24 \times 24$
same. To update a component in any block of the four points, its computational complexity (number of arithmetical operations) is equal to (3 multiplications ( $M$ ) +11 additions ( $A$ )), therefore for mesh size $(N \times N)$ there are $((3 M+11 A) . N)$ operations for each line in each subset. Thus for $P$ parallel paths and $N_{r}=N / P$ lines in each subset, the total number of operations carried out by each processor is equal to $T=\left((3 M+11 A) \cdot N \cdot N_{r}\right)$. Besides the computational time $T$ there are some delay times due to the overheads incurred by the system which may degrade the algorithm's performance. These overheads are the generation of the parallel paths and the synchronisation at the end of each iteration cycle. These overheads may become significant, for example, if we consider the implementation of the synchronous and asynchronous 4point block Version One natural ordering on the NEPTUNE system using four processors for $\omega=1.0$ and a mesh size ( $24 \times 24$ ). In the asynchronous algorithm the problem converges after 286 iterations using a total of four parallel paths, while in the synchronous algorithm the problem converges after 286 iterations using a total of 574 parallel paths, because we need a synchronisation after each iteration. Therefore, it is clear that the amount of overheads especially the synchronisation overhead in our case may affect the performance of an algorithm. Hence, for this reason, we can say that the use of the asynchronous strategy for the parallel 4-point block iterative method is better suited for MIMD computers.

Now we compare the results obtained from the implementation of both the parallel 4-point and 9-point block iterative methods on the NEPTUNE system. In all the implemented strategies (natural and red-black) and versions (Version One and Two) using the synchronous and asynchronous
schemes we have, for $\omega=1.0$ the timing results for the parallel 9-point block implementation are less than that of the parallel 4-point iterative method. While for $\omega=\omega_{\text {opt }}$ (optimal $\omega$ ) we have the timing results for the parallel 4 -point block iterative method is less than that of the parallel 9-point block iterative method. However, the number of iterations needed for the problem to converge using the parallel 9-point block iterative method is less than that needed for the parallel 4-point block iterative method for both $\omega=1.0$ and $\omega=\omega$ opt and the speedup ratios in both methods are almost the same.

The above argument can be shown for example, for the synchronous natural ordering using Version Two, mesh size $(24 \times 24)$, the number of parallel paths is equal to one for both $\omega=1.0$ and $\omega=\omega_{\text {opt }}$. For the parallel 9-point and 4-point block iterative methods the total time can be calculated from both the total computational operations (equations (4.10.10) and (4.11.28)) and the number of iterations carried out by each parallel path (Tables (4.7) and (4.13)). Thus, using the parallel 9-point block iterative method with $\omega=1.0$, and $N_{r}=N / P$, the total computational operations is equal to $\left(\left(\frac{38}{9} M+\frac{59}{9} A\right) \cdot N \cdot N_{r} \cdot 201\right)$ and by substituting the timing for the multiplication operation (M) and adđition operation (A) for the NEPTUNE system we get the total timing is equal to (1377676.11 N. $N_{r}$ ) $\mu$ secs. While for the parallel 4-point block iterative method with $\omega=1.0$, the total computational operations is equal to $\left(\left(\frac{5}{2} M+\frac{11}{2} A\right) \cdot N \cdot N_{r} \cdot 286\right)$ and by replacing the operational timing for $M$ and A, we find that the total timing for the parallel 4-point iterative method is equal to (1458314.N. $\mathrm{N}_{r}$ ) $\mu \mathrm{sec}$. By comparing these two total timings it is clear that the total time in the case of parallel 9-point iterative method is less than that of parallel 4-point block iterative method using $\omega=1.0$.

Now by taking $\omega=$ optimal value ( $\omega_{\text {opt }}$ ), we find that the total timing using the parallel 9-point block is equal to ( $\left.\left(\frac{38}{9} M+\frac{59}{9} A\right) \cdot N \cdot N_{r} \cdot 32\right)$ and for that of the parallel 4 -point block is equal to ( $\left.\left(\frac{5}{2} \mathrm{M}+\frac{11}{2} \mathrm{~A}\right) \cdot \mathrm{N} \cdot \mathrm{N}_{r} \cdot 39\right)$. By replacing the values of $M$ and $A$ in both cases we get the total timing for the parallel 9-point block iterative method equal to (219331.52 N. $\mathrm{N}_{\mathrm{r}}$ ) $\mu s e c s$, while for the parallel 4 -point block iterative method is equal to (198861 N. $\mathrm{N}_{r}$ ) $\mu \mathrm{sec}$. Also by comparing these two totals we find that the total timing results in the case of the parallel 4 -point block iterative method is less than that of the parallel 9-point block iterative method. So, it is clear that the results obtained in our calculations are generally coincident with those obtained experimentally. Besides the computational operations time calculated above there is an extra overhead time which we do not include in our calculation and will be discussed later.

The timing results of both the asynchronous Version One and Two methods using both the parallel 9-point block and parallel 4-point block iterative method with $\omega=1.0, \omega=\omega_{\text {opt }}$ and mesh size $(24 \times 24)$ are shown in Figure (4.30). The total number of arithmetical operations required for the solution of the model problem are shown in Table $4.14 a, b, c, d$, which are calculated from combining the total number of computational operations given in equations (4.10.9), (4.10.10), (4.11.22) and (4.11.28) with the number of iterations obtained from the experimental results shown in Tables $(4.2),(4.3),(4.4),(4.5),(4.6),(4.7),(4.8),(4.9)$, (4.10), (4.11), (4.12) and (4.13). In Table (4.4), K=N. $\mathrm{N}_{r}$, where N represents the number of points in the mesh to be solved, $N_{r}=N / P$ is the number of lines in each process and all the figures are taken to the nearest integer.


FIGURE 4.30

The timing results of natural asynchronous version one and two using parallel 9 -point and 4-point block iterative method using mesh size $24 \times 24$

$K=N N_{r}, \quad N_{r}=N / P$
FIGURE 4.14(a)

$\mathrm{K}=\mathrm{N} \cdot \mathrm{N}_{r}, \quad \mathrm{~N}_{r}=\mathrm{N} / \mathrm{P}$

| Mesh Size ( $\mathrm{N}^{\times} \mathrm{N}$ ) | P | $\omega$ | Asynch.4-point Natural Version 1 |  | Asynch.4-point Red-Black Version 1 |  | Synch.4-point Natural Version 1 |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  |  | M | A | M | A | M | A |
| (24×24) | 1 | 1.0 | 858 K | 1859 K | 870 K | 1885 K | 858 K | 1859 K |
|  | 2 | 1.0 | 864 K | 1872 K | 879 K | 1905 K | 855 K | 1853 K |
|  | 3 | 1.0 | 867 K | 1879 K | 879 K | 1905 K | 855 K | 1853 K |
|  | 4 | 1.0 | 867 K | 1879 K | 882 K | 1911 K | 852 K | 1846 K |
|  | 1 | $\omega_{\text {opt }}$ | 120 K | 260 K | 123 K | 267 K | 120 K | 260 K |
|  | 2 | $\omega_{\text {opt }}$ | 117 K | 254 K | 129 K | 280 K | 117 K | 254 K |
|  | 3 | $\omega_{\text {opt }}$ | 117 K | 254 K | 129 K | 280 K | 117 K | 254 K |
|  | 4 | $\omega_{\text {opt }}$ | 117 K | 254 K | 129 K | 280 K | 114 K | 247 K |

$\mathrm{K}=\mathrm{N}^{\star} \mathrm{N}_{r}, \quad \mathrm{~N}_{r}=\mathrm{N} / \mathrm{P}$
FIGURE 4.14 (c)

| Mesh Size ( $\mathrm{N}^{\times} \mathrm{N}$ ) | P | $\omega$ | Asynch.4-point Natural Version 2 |  | Asynch.4-point Red-Black Version 2 |  | Synch.4-point Natural Version 2 |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  |  | M | A | M | A | M | A |
| (24×24) | 1 | 1.0 | 715 K | 1573 K | 728 K | 1601 K | 715 K | 1573 K |
|  | 2 | 1.0 | 718 K | 1579 K | 733 K | 1612 K | 713 K | 1568 K |
|  | 3 | 1.0 | 720 K | 1584 K | 733 K | 1612 K | 710 K | 1562 K |
|  | 4 | 1.0 | 723 K | 1590 K | 735 K | 1617 K | 708 K | 1557 K |
|  | 1 | $\omega_{\text {opt }}$ | 98 K |  | 98 K | 215 K | 98 K | 215 K |
|  | 2 | $\omega$ opt | 95 K | 209 K | 100 K | 220 K | 95 K | 209 K |
|  | 3 | $\omega_{\text {opt }}$ | 93 K | 204 K | 100 K | 220 K | 93 K | 204 K |
|  | 4 | $\omega_{\text {opt }}$ | 93 K | 204 K | 98 K | 215 K | 90 K | 198 K |

$$
\mathrm{K}=\mathrm{N}^{*} \mathrm{~N}_{r}, \quad \mathrm{~N}_{r}=\mathrm{N} / \mathrm{P}
$$

Now, we discuss further the analysis of the parallel 9-point and parallel 4-point block iterative methods using the experimental results obtained when the algorithms were run on the NEPTUNE system and the resource timings of the NEPTUNE system are shown in Table (4.15). The principle behind the analysis of the two parallel block iterative methods is that parallel computing involves the sharing of some system resources which have a limited availability. Parallelism can often be introduced into a program or problem in a number of different demands for parallel resources. In reality parallel computers require the resources:
(i) Multiple processors,
(ii) Communication for data sharing,
(iii) Synchronisation to allow unique data modification. Three factors that affect the performance of parallel programs and are associated with the above three features of systems:
(a) The degree of parallelism in the program,
(b) Accesses to shared data space impose an overhead,
(c) Accesses to the synchronisation tool and their protected data structures impose an overhead.

Thus, since parallel programs always require more than one processor, there has to be some communication between the processors even if this is only as much as that required to start processing in the first instance. Parallel programs demand shared resources such as processors, memory block or a shared data structure and all resources demands by the algorithm directly affect the potential performance of the algorithm. The source of overheads on an asynchronous type machine can be classified into two types known as static and dynamic overheads [Barlow et al (1982)],

1. The static overheads due to the design of software and hardware. This covers the subdivision of the task, allocation of the tasks to the processors, checking by hardware and software for contention on accesses to the database, checking for correct sequencing.
2. The dynamic overheads due to the interference between two or more subtasks running on different processors and inevitably causing one or more of the processors to wait.

The performance of a multiprocessor can be expressed as a speedup factor ( $S_{p}$ ),

$$
S_{p}=\frac{T(1)}{T(p)}=\frac{\text { time taken on a single processor }}{\text { time taken on a } P \text { processor system }}
$$

or in terms of the time wasted (W),

$$
W=P * T(p)-T(1),
$$

where the wasted time is equal to the sum of times taken by the $p$ processors to complete their subtasks less the time taken on a uniprocessor. The time wasted must be equal to the sum of the static and dynamic overheads. It is obvious that either all processors complete processing together or some processors take longer than others thus,

$$
T(p) \geqslant \frac{T(1)+W}{P}
$$

it follows that

$$
S_{p} \leqslant \frac{P T(1)}{T(I)+W}
$$

Maximum speed-up factors ( $S_{p}$ ) can be obtained by assuming that the dynamic overheads are zero, and this is true only if every request for a resource occurs when that resource is not being used. This is clearly impossible if the demands for a resource are greater than the supply of that resource. The three resources which are required for parallel computing are processors, shared memory and synchronisation. The NEPTUNE
system has the ability to provide these resources,
(i) Processors

The software that controls the scheduling of processors to processes counts the number of processes run by each processor. This software also counts any time that a processor is idle because there are no ready processes to run.
(ii) Shared Memory

This can be measured by counting the number of accesses to a shared data by going through the user's program.
(iii) Synchronisation

The cost of mutual exclusion is significant because a high level software technique has to be used to overcome inadequacies in the hardware.

In order to carry out the performance analysis of all the implemented algorithms discussed in this section we need to know the resource times of the NEPTUNE system, which is obtained from Barlow et al [1981] and Woodward, et al [1983] and illustrated in Table (4.15). In our analysis of the parallel 9-point block and parallel 4-point block iterative methods we will concentrate on the natural ordering of both methods Version One and Two with mesh size $(24 \times 24), \omega=1.0$ and $\omega=\omega_{\text {opt }}$. When the NEPTUNE system is used the parallel control access overheads and the shared data access overheads will be calculated using the following commands:
(i) The $\operatorname{XPFCLS}\left(T_{S}\right)$ : this command generates a load module with no shared data assigned into the shared memory and no parallel path allocation. Therefore, all the parallel programming constructs will be treated as ordinary Fortran constructs.
(ii) The $\operatorname{XPFCLN}\left(\mathrm{T}_{\mathrm{N}}\right)$ : this command only loads the shared data into the shared memory. Thus, by comparing the result of this command with the results obtained from (i), we obtain the shared data access overheads.
(iii) The XPFCL $\left(T_{P}\right)$ : this command generates the load module as in (ii) in addition to the parallel path allocation. The comparison of these results with that of (ii) yields the parallel control overheads.

| Resource Processor | 0 | 1 | 2 | 3 |
| :---: | :---: | :---: | :---: | :---: |
| Relative Speeds | 1.000 | 1.014 | 1.006 | 1.019 |
| Memory access times |  |  |  |  |
| Local* | 0.98 | 0.95 | 0.92 | 0.92 |
| Shared* | (0.98+0.75) | (0.95+0.75) | (0.92+0.76) | (0.92+0.76) |
| Mutual exclusion mechanism* | $\sim 800$ | $\sim 800$ | ~800 | ~800 |
| Mutual exclusion blocked* | $\sim 400$ | $\sim 400$ | $\sim 400$ | $\sim 400$ |
| Parallel path overhead* | $\sim 1200$ | $\sim 1200$ | ~1200 | $\sim 1200$ |
| Floating point* | $\sim 720$ | $\sim 720$ | $\sim 720$ | ~720 |
| Integer* | $\sim 20$ | $\sim 20$ | $\sim 20$ | $\sim 20$ |

*times in microseconds.
TABLE 4.15: The resources time on the NEPTUNE system

The resource demands required by Version One and Two of the parallel 9-point and parallel 4-point block iterative methods using the synchronous and asynchronous strategy with natural ordering are shown in Table 4.16. Actually Table 4.16 gives the mean rate of access to shared data and parallel paths. It gives the estimates of the potential speed-up from using $P$ processors, where $N$ represents the number of rows in the mesh to
be solved. While, Table 4.17 illustrates the results obtained when the algorithms were run on the NEPTUNE system using mesh size (24×24), where the parallel control overhead (PCO) is calculated using the formula $P C O=\left(T_{P}-T_{N}\right) / T_{P} * 100$, and shared data overhead (SDO) is calculated using the formula $S D O=\left(T_{N}-T_{S}\right) / T_{P} * 100$. The term "flops" in these tables represents a floating point operation.

Now from inspection of the parallel 9-point and parallel 4-point iterative methods we see that a linear speed-up has been achieved and up to the number $(N / 3)^{2}$ processors can be employed as an upper limit in the case of the parallel 9-point block iterative method and (N/2) ${ }^{2}$ processors in the case of the parallel 4-point block iterative method. As an example, in the case of the parallel 9-point asynchronous Version One using natural ordering of the points, the algorithm has made 42 accesses to the shared data per 154 floating point operations. From the results of Table 4.15 the shared data access timing is~0.75 usecs and $\sim 720 \mu \operatorname{secs}$ for a floating point operation. Therefore, the static shared data access overheads in this algorithm is obtained as follows:

$$
\frac{1}{\left(\frac{154}{42}\right)} * \frac{0.75}{720} * 100=0.028 \%
$$

Further the parallel path access loss is equal to $0.01 \%$ since the program made 1 access per ( $16 * N^{*} N_{r}$ ) flops and the parallel path mechanism requires $\sim 1200$ microseconds (see Table 4.15). From these two tables it can be concluded that the experimental results obtained from the NEPTUNE system and the predicted results obtained from accesses to the shared data or a parallel path are in agreement. Also, from the figures in Tables 4.16 and 4.17 it is clear that the losses using the parallel 9-point block iterative method are less than that of using the parallel

4-point block iterative method. This is mainly because the number of blocks in the parallel 4-point block iterative method is greater than that in the parallel 9-point block iterative method and the number of iterations needed for the problem to converge using the parallel 4-point block iteration method is greater than that using the parallel 9-point block iterative method. Hence, more losses will occur when using the parallel 4-point block iterative methods.

| Program | Processors (P) |  | Shared Data |  | Parallel Path |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Numbers | Speed-up | Access Rate | Overhead amount | Access Rate | Overhead amount |
| 9-point Asynchronous Version One, Natural Ordering | $\mathrm{P} \leqslant \mathrm{N} / 3$ | O(P) | 42:154 flops | 0.028\% | $\begin{aligned} & 1:\left(16 \star N^{\star} N_{r}\right) \\ & \text { flops } \end{aligned}$ | 0.018\% |
| 9-point Synchronous Version One, Natural Ordering | $\mathrm{P} \leqslant \mathrm{N} / 3$ | O(P) | 42:154 flops | 0.028\% | $\begin{aligned} & \text { 1: }\left(16 \star N^{*} N_{r}\right) \\ & \text { flops } \end{aligned}$ | 0.018\% |
| 9-point Asynchronous Version Two, Natural Ordering | $\mathrm{P} \leqslant \mathrm{N} / 3$ | O(P) | 34:129 flops | 0.028\% | $\begin{aligned} & 1:\left(10.78 * N^{*}\right. \\ & \left.N_{r} \text { flops }\right) \end{aligned}$ | $0.027 \%$ |
| 9-point Synchronous Version Two, Natural Ordering | $\mathrm{P} \leqslant \mathrm{N} / 3$ | O(P) | 34:129 flops | 0.028\% | $\begin{aligned} & 1:\left(10.78 * N^{*}\right. \\ & N_{r} \text { flops) } \end{aligned}$ | $0.027 \%$ |
| 4-point Asynchronous Version One, Natural Ordering | $\mathrm{P} \leqslant \mathrm{N} / 2$ | O(P) | 16:50 flops | 0.033\% | $\begin{aligned} & \text { l: }\left(9.5 * N^{*} N_{r}\right) \\ & \text { flops } \end{aligned}$ | 0.031\% |
| 4-point Synchronous Version One, Natural Ordering | $\mathrm{P} \leqslant \mathrm{N} / 2$ | O(P) | 16:50 flops | 0.033\% | $\begin{aligned} & \text { l: }\left(9.5 * N_{r}^{*} N_{r}\right) \\ & \text { flops } \end{aligned}$ | 0.031\% |
| 4-point Asynchronous Version Two, Natural Ordering | $\mathrm{P} \leqslant \mathrm{N} / 2$ | O(P) | 16:44 flops | 0.038\% | $\begin{aligned} & 1:\left(8{ }^{*} N^{*} N_{r}\right) \\ & \text { flops } \end{aligned}$ | $0.036 \%$ |
| 4-point Synchronous Version Two, Natural Ordering | $\mathrm{P} \leqslant \mathrm{N} / 2$ | O(P) | 16:44 flops | 0.038\% | $\begin{aligned} & 1:\left(8^{*} N^{*} N_{r}\right) \\ & \text { flops } \end{aligned}$ | 0.036\% |


| Program | $\omega$ | Speed-up |  |  | Shared Data Overhead (SDO) | Parallel Control Overhead (PCO) |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | 2 | 3 | 4 |  |  |
| 9-point Asynchronous | 1.0 | 1.97826 | - | 3.94090 | . 0288 | .019\% |
| Version One | $\omega_{\text {opt }}$ | 1.99154 | - | 3.99252 | .031\% | . 0198 |
| 9-point Synchronous | 1.0 | 1.97520 | - | 3.93692 | . $030 \%$ | . 0198 |
| Version One | $\omega_{\text {opt }}$ | 1.98813 | - | 3.93152 | . $031 \%$ | . 0198 |
| 9-point Asynchronous | 1.0 | 1.98204 | - | 3.96592 | . 0298 | . 0288 |
| Version Two | $\omega_{\text {opt }}$ | 1.97627 | - | 3.95138 | .030\% | . $030 \%$ |
| 9-point Synchronous | 1.0 | 1.96687 | - | 3.95216 | . 0298 | .028\% |
| Version Two | $\omega_{\text {opt }}$ | 1.94355 | - | 3.94975 | . $029 \%$ | .029\% |
| 4-point Asynchronous | 1.0 | 1.98707 | 2.96230 | 3.96828 | . 0348 | . 0348 |
| Version One | $\omega_{\text {opt }}$ | 1.99417 | 2.98333 | 3.99575 | .033\% | .033\% |
| 4-point Synchronous | 1.0 | 1.97248 | 2.93550 | 3.91340 | . 0338 | . 0348 |
| Version One | $\omega_{\mathrm{opt}}$ | 1.98774 | 2.98180 | 3.99073 | .033\% | .033\% |
| 4-point Asynchronous | 1.0 | 1.98571 | 2.97185 | 3.94873 | . $040 \%$ | . 0348 |
| Version Two | $\omega_{\text {opt }}$ | 1.98985 | 2.99889 | 3.97560 | .037\% | .037\% |
| 4-point Synchronous | 1.0 | 1.97494 | 2.94722 | 3.91911 | . 0408 | . $038 \%$ |
| Version Two | $\omega_{\text {opt }}$ | 1.98010 | 2.98830 | 3.95963 | .037\% | .044\% |

TABLE 4.17: Performance measurements of algorithms on the NEPTUNE system for mesh size ( $24 \times 24$ ) using natural ordering strategy

## CONCLUSIONS

From the basic concept of the 9 -point explicit block iterative method the parallel 9-point block iterative method was developed and implemented on the NEPTUNE system. The implementation of the parallel 9-point and parallel 4-point block iterative methods were programmed using different versions and strategies such as synchronous and asynchronous together with natural or red-black ordering. It is clear that the implementation of different strategies present different timing results and losses when they are run on the NEPTUNE system. In the two implemented versions of both parallel 9-point and parallel 4point block iterative method, Version Two gives better timing results in all the programmed strategies, and this is due to the way in which each block within each subset is evaluated.

The asynchronous natural ordering strategy always gives better results than that using other strategies. For example, if the results obtained from that strategy are compared with the corresponding results obtained from the synchronous natural ordering strategy, the asynchronous strategy gives better results than that of the synchronous version due to the synchronisation overheads needed at the end of each iteration. Also, in the asynchronous implementation better results are obtained because the processors were almost always fully occupied and busy doing work most of the time.

In general, all the algorithms presented in this chapter have predicted static parallel path access overheads and shared data access overheads which agree with the overheads obtained when these algorithms were run on the NEPTUNE system.

It can be seen from the experimental results that the shared data access overhead and the parallel control access overhead in the case of the parallel 9-point block iterative methods are less than that of the parallel 4-point block iterative method. Also, the parallel 9-point block iterative methods take less time than its corresponding parallel 4-point block iterative method when $\omega=1.0$, while when $\omega=\omega_{\text {opt' }}$ in general the two parallel methods take the same time. Therefore, the parallel 9-point block iterative method was chosen as best amongst the two parallel block methods.

Finally to conclude this chapter it can be seen that these two parallel block methods are suited for the parallel implementation on the MIMD computer and this is due to the almost linear speed-ups obtained from their implementations.

## Chapter Five

## THE PARALLEL ALTERNATING GROUP EXPLICIT

(A,G.E.) METHOD

### 5.1 INTRODUCTION

Point (Explicit) methods have natural extensions to block iterative processes in which groups of components of $\mathbf{x}^{(k)}$ are modified simultaneously (see Chapter 4). This will involve the simultaneous solution of a system of linear equations. Consequently, individual components are implicitly defined in terms of other components of the same group or block. Such a method is called an implicit iterative or block iterative method. The redefinition of an explicit method so that it becomes implicit often leads to an increase in the convergence rate at the cost of some complication in the computational algorithm. The blocks may be a single row of points, two rows, etc. A better convergence rate can be obtained by evaluatine the wholé line, as an example the line S.O.R. gives an improvement by a factor of $\sqrt{2}$ over the corresponding optimum S.O.R. by points (Parter [1961]).

In this chapter we discuss a class of methods for solving a twopoint boundary value problem. These methods are the Alternating Direction Implicit (A.D.I.) Method and the Alternating Group Explicit (A.G.E.) Method.

Two parallel strategies of the A.G.E. method were developed and implemented on the NEPTUNE system. These include synchronous and asynchronous versions of the algorithms. The parallel A.G.E. method was used to solve a one dimensional linear and non-jinear boundary value problem. The results from these implementations were compared as well as the performance, analysis of the best method presented. Also, the timing results from the parallel A.G.E. method implementation were compared with the parallel Jacobi, Gauss-Seidel and S.O.R. methods.

Consider the differential equation,

$$
\begin{equation*}
-\frac{d^{2} u}{d x^{2}}+q(x) u=f(x) \tag{5.1.1}
\end{equation*}
$$

subject to the two-point boundary conditions,

$$
\begin{equation*}
U(a)=\alpha, \quad U(b)=\beta . \tag{5.1.2}
\end{equation*}
$$

Here $\alpha$ and $\beta$ are given real constants, and $f(x)$ and $q(x)$ are given real continuous functions in $a \leqslant x \leqslant b$, with, $q(x) \geqslant 0$. For simplicity, we place a uniform mesh of size $h$, where,

$$
\begin{equation*}
h=(b-a) /(N+1) \tag{5.1,3}
\end{equation*}
$$

on the interval $a \leqslant x \leqslant b$, and we denote the mesh points of the discrete problem by,

$$
\begin{equation*}
x_{i}=a+i h, \quad 0 \leqslant i \leqslant N+1 \tag{5.1.4}
\end{equation*}
$$

as illustrated in the Figure 5.1,


FIGURE 5.1

The finite difference method (Chapter 4) is used to derive a finite difference approximation to (5.1.1) by using a finite Taylor's series expansion of the solution $U(x)$ to (5.1.1). Let us assume that the (unique) solution $U(x)$ of (5.1.1) is of class $c^{4}$ in $a \leq x \leq b$, i.e. $d^{2} u / d x^{2}$ exists and is continuous in this interval. Denoting $U\left(X_{i}\right)$ by $U_{i}$, the finite Taylor expansion of $U_{i+1}$ is,

$$
\begin{equation*}
U_{i \pm 1}=U_{i} \pm h\left(\frac{d U_{i}}{d x}\right)+\frac{h^{2}}{2!}\left(\frac{d^{2} U_{i}^{1}}{d x^{2}}\right) \pm \frac{h^{3}}{3!}\left(\frac{d^{3} U_{i}}{d x^{3}}\right)+\frac{h^{4}}{4!} \frac{d^{4}}{d x^{2}} \pm \ldots \tag{5.1.5}
\end{equation*}
$$

from which it follows that,

$$
\begin{equation*}
-\frac{d^{2} U_{i}}{d x^{2}}=\frac{2 U_{i}-U_{i-1}-U_{i+1}}{h^{2}}+\frac{h^{2}}{12} \frac{d^{4} U_{i}}{d x^{4}}+\ldots \tag{5.1.6}
\end{equation*}
$$

Therefore, by substitution in (5.1.1) gives,

$$
\begin{equation*}
\frac{1}{h^{2}}\left(2 u_{i}-u_{i+1}-u_{i-1}\right)+q_{i} u_{i}=f_{i}, 1 \leqslant i \leqslant N, \tag{5.1.7}
\end{equation*}
$$

with a local truncation error of

$$
\frac{h^{2}}{12} \frac{d^{4} v_{i}}{d x^{4}}+\ldots
$$

where $u_{i}$ denotes the function satisfying the difference equation at the mesh point $x_{i}=a+i h$.

Since $U_{O}=\alpha$ and $U_{N+1}=\beta$, then we have $N$ equations for the $N$ unknowns $u_{i}, i=1,2, \ldots, N$. In the matrix notation, (5.1.7) can be written in the form,

$$
\begin{equation*}
\mathrm{A} \underline{\mathrm{u}}=\underline{\mathrm{b}}, \tag{5.1.8}
\end{equation*}
$$

where $A$ is a real ( $N^{x} N$ ) matrix, $u$ is the discrete approximation vector to the solution $U(x)$ of (5.1.1)-(5.1.2) and $\underline{b}$ is a column vector given by,

$$
\begin{align*}
& \underline{u}=\left[\begin{array}{c}
u_{1} \\
u_{2} \\
1 \\
1 \\
1 \\
1 \\
u_{N-1} \\
u_{N}
\end{array}\right] \text {. and } \underline{b}=h^{2}\left[\begin{array}{c}
f_{1}+\alpha / h^{2} \\
f_{2} \\
1 \\
1 \\
1 \\
f_{N-1} \\
f_{N}+B / h^{2}
\end{array}\right] \tag{5.1.9b}
\end{align*}
$$

The basic properties of the matrix A are real, symmetric and tridiagonal and since $q(x) \geqslant 0$ then it is also diagonally dominant with positive diagonal entries. From the directed graph of $A$ it can be seen that
since it is strongly connected implies that $A$ is also irreducible.
The A.D.I. and A.G.E. methods for solving the system (5.1.8) which is based on splitting the matrix $A$ of (5.1.9a) into ( $N \times N$ ) matrices as shown in the next two sections.

### 5.2 ALTERNATING DIRECTION IMPLICIT (A.D.I.) METHODS

The S.O.R. (Chapter 4) method by lines proceeds by taking all the lines in the same direction. Thus in Figure (5.2), for example, we first solve for the values at $1,2,3$, then for $4,5,6$, and finally for 7,8,9. Then we begin again with $1,2,3$, and so forth. Convergence is often improved by following the first sequence with a second in the column direction. Thus, a complete iteration consists of first a half iteration in the row direction followed by a second half iteration in the column direction. Such methods are aptly designated altermating direction implicit methods or A.D.I. methods for short.


FIGURE 5.2

Varga [1962] shows that, the first A.D.I. methods were developed by Peaceman and Rachford [1955] for solving the system

$$
\begin{equation*}
A \underline{u}=\underline{b} . \tag{5.2.1}
\end{equation*}
$$

where the $(n \times n)$ matrix $A$ is an irreducible Stieltjes matrix, i.e. A is a real symmetric and positive irreducible matrix with non-positive off-diagonal entries. Moreover, that the matrix $A$ has at most five nonzero entries in any of its rows. Matrix A is non-singular and can be represented as the sum of three ( $n \times n$ ) matrices,

$$
\begin{equation*}
A=H+V+\Sigma . \tag{5.2.2}
\end{equation*}
$$

The matrices $H, V$ and $\Sigma$ are all real symmetric ( $n \times n$ ) matrices. The matrix $\Sigma$ is a non-negative diagonal matrix, and is thus non-negative definite. The matrices $H$ and $V$ are associated with the finite difference approximation to the partial differential equation in the $x$ and $y$ directions respectively and each have no more than three non-zero entries per row, and both $H$ and $V$ are diagonally dominant matrices with positive diagonal entries and non-positive off-diagonal entries, and are thus Stieltjes matrices.

By using (5.2.2) we can write the matrix equation (5.2.1) as a pair of matrix equations,

$$
\left.\begin{array}{l}
\left(H+\frac{1}{2} \Sigma+r I\right) \underline{u}=\underline{b}-\left(V+\frac{1}{2} \Sigma-r I\right) \underline{u},  \tag{5.2.3}\\
\left(V+\frac{1}{2} \Sigma+r I\right) \underline{u}=\underline{b}-\left(H+\frac{1}{2} \Sigma-r I\right) \underline{u} .
\end{array}\right\}
$$

for any positive scalar r. If we let,

$$
\begin{equation*}
H_{1}=H+\frac{1}{2} \Sigma, \quad V_{1}=V+\frac{1}{2} \Sigma \text {, } \tag{5.2.4}
\end{equation*}
$$

then the Peaceman-Rachford altermating-direction implicit method is defined by

$$
\left.\begin{array}{l}
\left(\mathrm{H}_{1}+r_{k+1} I\right) \underline{u}^{\left(k+\frac{1}{2}\right)}=\underline{b}-\left(V_{1}-r_{k+1} I\right) \underline{u}^{(k)},  \tag{5.2.5}\\
\left(V_{1}+r_{k+1} I\right) \underline{u}^{(k+1)}=\underline{b}-\left(H_{1}-r_{k+1} I\right) \underline{u}^{\left(k+\frac{1}{2}\right)},
\end{array}\right\}
$$

where the r's are positive acceleration parameters chosen to make the process converge more rapidly, and $\underline{u}^{(0)}$ is an arbitrary initial vector approximation to the unique solution of (5.2.1).

Since the matrices $\left(H_{1}+r_{k+1} I\right)$ and $\left(V_{1}+r_{k+1} I\right)$ are, after suitable permutations, tridiagonal nonsingular matrices, the above implicit process can be directly carried out by the simple algorithm based on the Gaussian elimination. In other words, this iterative method can be thought of as
a line method with alternating directions. Indeed, the name alternatingdirection method is derived from the observation that for the first equation (5.2.5) we solve first along horizontal mesh lines, and then for the second equation of (5.2.5), we solve along vertical mesh lines. The vector $\underline{u}^{\left(k+\frac{1}{2}\right)}$ is treated as an auxiliary vector which is, discarded as soon as it has been used in the calculation of $u^{(k+1)}$ The two equations of (5.2.5) are now combined to give,

$$
\begin{equation*}
\underline{u}^{(k+1)}=T_{r_{k+1}} \underline{u}^{(k)}+g_{r_{k+1}}(\underline{b}), k \geqslant 0, \tag{5.2.6}
\end{equation*}
$$

where,

$$
\begin{equation*}
T_{r}=\left(V_{1}+r I\right)^{-1}\left(H_{1}-r I\right)\left(H_{1}+r I\right)^{-1}\left(V_{1}-r I\right) \tag{5,2.7a}
\end{equation*}
$$

and

$$
g_{r}(b)=\left(V_{1}+r I\right)^{-1}\left\{I-\left(H_{1}-r I\right)\left(H_{1}+r I\right)^{-1}\right\}_{\underline{b}} .
$$

It will be noticed that (5.2.6) is of the same form as (5.2.3).
For the convergence of the A.D.I. method Varga [1962] states the following theorem.

## Theorem 5.1

Let $H_{1}$ and $V_{1}$ be $(n \times n)$ symmetric non-negative definite matrices, where at least one of the matrices $H_{1}$ and $V_{1}$ is positive definite. Then, for any $r>0$, the Peaceman-Rachford matrix $T_{r}$ of (5.2.7a) is convergent.

## Proof:

See Varga [1962], page 123.

### 5.3 THE ALTERNATING GROUP EXPLICIT (A.G.E.) METHOD

Evans [1985] presents a class of methods for solving (5.1.8) which is based on the "splitting" of the matrix $A$ into the sum of three matrices,

$$
\begin{equation*}
A=G_{1}+G_{2}+\Sigma \tag{5.3.1}
\end{equation*}
$$

where $\Sigma$ is a non-negative diagonal matrix and where $G_{1}, G_{2}$ and $\Sigma$ satisfy the following conditions:
(i) $G_{1}+\theta \Sigma+r I$ and $G_{2}+\theta \Sigma+r I$ are non-singular for any $\theta \geqslant 0, r>0$.
(ii) for any vector $\underline{v}_{1}$ and $\underline{v}_{2}$ and for any constants $\theta \geqslant 0$ and $r>0$ it is "convenient" to solve the systems explicitly, i.e.,

$$
\underline{Y}_{1}=G_{1}^{-1} \underline{v}_{1} \quad \text { and } \quad \underline{Y}_{2}=G_{2}^{-1} \underline{v}_{2}
$$

for $\underline{Y}_{1}$ and $\underline{Y}_{2}$ respectively.
We shall be concerned here with the situation where $G_{1}$ and $G_{2}$ are either small $2 \times 2$ block systems or can be made so by a suitable permutation of their rows and corresponding columns. This procedure is "convenient" in the sense that the work required is much less than would be required to solve the original system (5.1.8) directly.

From the above discussion, we have $G_{1}, G_{2}$ and $\Sigma$ given by,

if $N$ is even and

if $n$ is odd, and,


Let us write (5.1.8) in the form,

$$
\begin{equation*}
\left(\mathrm{G}_{1}+\mathrm{G}_{2}+\Sigma\right) \underline{\mathrm{u}}=\underline{\mathrm{b}} \tag{5,3.3}
\end{equation*}
$$

and let us consider two equivalent forms,

$$
\begin{equation*}
\left(G_{1}+\theta \Sigma+r I\right) \underline{u}=\underline{b}-\left(G_{2}+(1-\theta) \Sigma-r I\right) \underline{u} . \tag{5,3.4}
\end{equation*}
$$

and

$$
\begin{equation*}
\left(G_{2}+\theta \Sigma+r^{\prime} I\right) \underline{\underline{u}}=\underline{b}-\left(G_{1}+(1-\theta) \Sigma-r^{\prime} I\right) \underline{u} . \tag{5.3.5}
\end{equation*}
$$

Analogous to the Peaceman-Rachford A.D.I. method [1955] one selects positive iteration parameters $r$ and $r^{\prime}$ and determines $\underline{u}^{\left(k+\frac{1}{2}\right)}$ and $\underline{u}^{(k+1)}$ respectively by,
and

$$
\begin{equation*}
\left(G_{1}+\theta \Sigma+r I\right) \underline{u}^{\left(k+\frac{1}{2}\right)}=\underline{\mathrm{b}}-\left(\mathrm{G}_{2}+(1-\theta) \Sigma-r I\right) \underline{\mathrm{u}}^{(k)}, \tag{5.3.6}
\end{equation*}
$$

where $\underline{u}^{0}$ is an arbitrary initial vector approximation of the unique solution of (5.1.8).

For simplicity, we shall consider here the special case where,

$$
\begin{equation*}
\theta=\hat{\theta}=\frac{1}{2}, r=r^{\prime}, \tag{5.3.8}
\end{equation*}
$$

and we let,

$$
\begin{equation*}
\bar{G}_{1}=G_{1}+\frac{1}{2}, \quad \bar{G}_{2}=G_{2}+\frac{1}{2} \Sigma \tag{5.3.9}
\end{equation*}
$$

Evidently $\bar{G}_{1}$ and $\bar{G}_{2}$ satisfy the following conditions:
(i) $\bar{G}_{1}+r I$ and $\bar{G}_{2}+r I$ are non-singular for any $r>0$,
(ii) for any vectors $\underline{v}_{1}$ and $\underline{v}_{2}$ and for any $r>0$ it is easy to solve the ( $2 \times 2$ ) systems explicitly,

$$
\begin{equation*}
\left(\bar{G}_{1}+r I\right) Y_{1}=\underline{v}_{1}, \quad\left(\bar{G}_{2}+r I\right) Y_{2}=\underline{v}_{2} . \tag{5.3.10}
\end{equation*}
$$

Therefore (5.3.3) becomes,

$$
\begin{equation*}
\left(\bar{G}_{1}+\bar{G}_{2}\right) \underline{u}=\underline{b} \tag{5.3.11}
\end{equation*}
$$

and (5.3.4)-(5.3.5) becomes respectively,

$$
\begin{align*}
& \left(\bar{G}_{1}+r I\right) \underline{u}^{\left(k+\frac{1}{2}\right)}=\underline{b}-\left(\bar{G}_{2}-r I\right) \underline{u}^{(k)}  \tag{5.3.12}\\
& \left(\bar{G}_{2}+r I\right) \underline{u}^{(k+1)}=\underline{b}-\left(\bar{G}_{1}-r I\right) \underline{u}^{\left(k+\frac{1}{2}\right)} \tag{5.3.13}
\end{align*}
$$

If we combine the above two equations into the form,

$$
\begin{equation*}
\underline{u}^{(k+1)}=T_{r} \underline{u}^{(k)}+\underline{b}_{\underline{r}} . \tag{5.3.14}
\end{equation*}
$$

where,

$$
\begin{equation*}
\underset{r}{\mathrm{~b}}=\left(\overline{\mathrm{G}}_{2}+r I\right)^{-1}\left\{I-\left({\overline{G_{1}}}_{1}-r I\right)\left({\overline{G_{1}}}_{1}+r I\right)^{-1}\right\} \underline{b}, \tag{5.3.15}
\end{equation*}
$$

and

$$
\begin{equation*}
T_{r}=\left(\bar{G}_{2}+r I\right)^{-1}\left(\bar{G}_{1}-r I\right)\left(\bar{G}_{1}+r I\right)^{-1}\left(G_{2}-r I\right) \tag{5.3.16}
\end{equation*}
$$

The matrix $T_{r}$ is called the A.G.E. iteration matrix.
To analyse the convergence of the A.G.E. method, we assume that $\underline{U}$
is the true solution of (5.1.8) then,

$$
\begin{equation*}
\left(\overline{\mathrm{G}}_{1}+\overline{\mathrm{G}}_{2}\right) \underline{\mathrm{U}}=\underline{\mathrm{b}}, \tag{5,3.17}
\end{equation*}
$$

and

$$
\begin{equation*}
\left(\bar{G}_{1}+r I\right) \underline{U}=\underline{b}-\left(\bar{G}_{2}-r I\right) \underline{U} . \tag{5.3.18}
\end{equation*}
$$

$$
\text { If } \underline{e}^{(k)}=\underline{u}^{(k)}-\underline{U} \text { is the error vector associated with the vector }
$$

iterate $\underline{u}^{(k)}$. Thus from (5.3.12) and (5.3.18) we have,

$$
\begin{equation*}
\left(\bar{G}_{1}+r I\right) e^{\left(k+\frac{1}{2}\right)}=-\left(\bar{G}_{2}-r I\right) e^{(k)} . \tag{5.3.19}
\end{equation*}
$$

Similarly,

$$
\begin{equation*}
\left(\bar{G}_{2}+r I\right) e^{(k+1)}=-\left(\bar{G}_{1}-r I\right) e^{\left(k+\frac{1}{2}\right)} \tag{5.3.20}
\end{equation*}
$$

and hence,

$$
\begin{equation*}
\underline{e}^{(k+1)}=T_{r} \underline{e}^{(k)} \tag{5.3.21}
\end{equation*}
$$

where $T_{r}$ is given in (5.3.16).
To show the convergence properties of $T_{r}$, we state the following theorem due to Evans [1985].

Theorem 5.2
If $\bar{G}_{1}$ and $\bar{G}_{2}$ are real positive definite matrices and if r>0 then $\rho\left(T_{r}\right)<I$.

Proof:
See Evans [1985].

Let us now assume that $\bar{G}_{1}$ and $\bar{G}_{2}$ are real positive definite matrices and that the eigenvalues $\lambda$ of $\bar{G}_{1}$ and $\mu$ of $\bar{G}_{2}$ lie in the ranges,

$$
\begin{equation*}
0 \leqslant a \leqslant \lambda \leqslant b, \quad 0 \leqslant a \leqslant \mu \leqslant b \tag{5.3.22}
\end{equation*}
$$

Evidently, if $r>0$ we have,

$$
\begin{align*}
\rho\left(T_{r}\right) & \leqslant \rho\left(\left(\bar{G}_{1}-r I\right)\left(\bar{G}_{1}+r I\right)^{-1}\right) \rho\left(\left(\bar{G}_{2}-r I\right)\left(\bar{G}_{2}+R I\right)^{-1}\right) \\
& \left.=\underset{a \leqslant \lambda \leqslant b}{\left(\max ^{\operatorname{lax}}\right.}\left|\frac{\lambda-r}{\lambda+r}\right|\right)\left(\underset{a \leqslant \mu \leqslant b}{ }\left|\frac{\mu-r}{\mu+r}\right|\right) \\
& =\underset{a \leq \gamma \leqslant b}{\left[\max ^{a}\left|\frac{\gamma-r}{\gamma+r}\right|\right]^{2}=\phi(a, b ; r) .} \tag{5.3.23}
\end{align*}
$$

Since $(\gamma-r) /(\gamma+r)$ is an increasing function of $\gamma$ we have,

$$
\begin{equation*}
\max _{a \leqslant \gamma \leqslant b}\left|\frac{\gamma-r}{\gamma+r}\right|=\max \left(\left|\frac{a-r}{a+r}\right| \cdot\left|\frac{b-r}{b+r}\right|\right) \tag{5.3.24}
\end{equation*}
$$

When $r=\sqrt{a b}$, then

$$
\begin{equation*}
\left|\frac{a-r}{a+r}\right|=\left|\frac{b-r}{b+r}\right|=\frac{\sqrt{b}-\sqrt{a}}{\sqrt{b}+\sqrt{a}} \tag{5,3.25}
\end{equation*}
$$

Moreover, if $0<r<\sqrt{a b}$ we have,

$$
\begin{equation*}
\left|\frac{b-r}{b+r}\right|-\frac{\sqrt{b}-\sqrt{a}}{\sqrt{b}+\sqrt{a}}=\frac{2 \sqrt{b}(\sqrt{a b-r})}{(b+r)(\sqrt{b}+\sqrt{a})}>0 \tag{5.3.26}
\end{equation*}
$$

and if $\sqrt{a b}<r$, then,

$$
\begin{equation*}
\left|\frac{a-r}{a+r}\right|-\frac{\sqrt{b} \sqrt{a}}{\sqrt{b}+\sqrt{a}}=\frac{2 \sqrt{b}(r \sqrt{a b})}{(r+a)(\sqrt{b}+\sqrt{a})}>0 . \tag{5.3.27}
\end{equation*}
$$

Thus $\phi(a, b ; r)$ is minimized when $r=\sqrt{a b}$ and

$$
\begin{equation*}
(T \sqrt{a b}) \leqslant \phi(a, b ; \sqrt{a b})=\left(\frac{\sqrt{b}-\sqrt{a}}{\sqrt{b}+\sqrt{a}}\right)^{2} \tag{5,3.28}
\end{equation*}
$$

Therefore, $r=\sqrt{a b}$ is optimum in the sense that the bound $\phi(a, b ; r)$ for $\rho\left(T_{r}\right)$ is minimized.

The convergence of the A.G.E. method is frequently very rapid if one allows $r$ to vary cyclically from iteration to iteration. This rapid convergence can be proved to hold for an appropriate choice of the iteration parameters in the commutative case. Following Birkoff, Varga and Young [1962], we say that the commutative case holds if the matrices $G_{1}, G_{2}$ and $\Sigma$ of (5.3.3) satisfy the following conditions:
(i) $G_{1} G_{2}=G_{2} G_{1}$.
(ii) $\Sigma=\sigma I$, where $\Sigma$ is a non-negative constant,
(iii) $G_{1}$ and $G_{2}$ are similar to non-negative diagonal matrices

If these assumptions hold then the matrices,
$\vec{G}_{1}=G_{1}+\frac{1}{2} \Sigma, \quad \vec{G}_{2}=G_{2}+\frac{1}{2} \Sigma$, satisfy the conditions,
(i) $\bar{G}_{1} \bar{G}_{2}=\bar{G}_{2} \bar{G}_{1}$,
(ii) $\bar{G}_{1}$ and $\bar{G}_{2}$ are similar to non-negative diagonal matrices

The importance of the above conditions depends on the following theorem of Frobenius, which we state without proof.

Theorem 5.3
If $\bar{G}_{1}$ and $\bar{G}_{2}$ are similar to diagonal matrices and if $\bar{G}_{1} \bar{G}_{2} \approx \bar{G}_{2} \bar{G}_{1}$, then there exists a non-singular matrix $W$ such that

$$
\begin{equation*}
W^{-1} \bar{G}_{1} W=D_{1}, \quad W^{-1} \bar{G}_{2} W=D_{2} \tag{5.3.34}
\end{equation*}
$$

where $D_{1}$ and $D_{2}$ are diagonal matrices.

It follows from (5.3.22) and (5.3.33) that there exists a set of linearly independent vectors $\underline{v}_{1}, \underline{v}_{2}, \ldots, \mathrm{v}_{\mathrm{k}}$, which corresponds to the columns of $W$ in Theorem (5.3), such that each $\underline{v}_{i}$ is an eigenvector both of $\bar{G}_{1}$ and of $\bar{G}_{2}$. Now, for any column $v$ of $W$ we have,

$$
\begin{equation*}
\bar{G}_{1} \underline{v}=\lambda \underline{v}, \quad \bar{G}_{2} \underline{v}=\mu \underline{v} \tag{5.3.35}
\end{equation*}
$$

for some eigenvalues $\lambda$ and $\mu$ of $\bar{G}_{1}$ and $\bar{G}_{2}$ respectively. Evidently,

$$
T_{r} \underline{V}=\left(\bar{G}_{2}+r I\right)^{-1}\left(\bar{G}_{1}-r I\right)\left(\bar{G}_{1}+r I\right)^{-1}\left(\bar{G}_{2}-r I\right) \underline{V}
$$

Since $\left(\bar{G}_{1}+r I\right)^{-1} \underline{v}=(\lambda+r)^{-1} \underline{v},\left(\bar{G}_{2}+r I\right)^{-1} \underline{v}=(\mu+r)^{-1} \underline{v}$ it follows that,

$$
T_{r} \underline{v}=\frac{(\lambda-r)(\mu-r)}{(\lambda+r)(\mu+r)} \underline{v}
$$

Thus $\underline{v}$ is an eigenvector of $T_{r}$ for any $r$.

### 5.4 EKPERIMENTAL RESULTS OF THE PARALLEL A.G.E. METHOD

The A.G.E. method was implemented in the parallel form where two strategies are used to solve a linear and a non-linear boundary value problem. In the two strategies, generally the problems are solved by decomposing it into many subsets that are assigned to the different processors which can then be run in parallel, i.e. the sequential decomposition is used (see Chapter 4). Also the two strategies are programmed on the NEPTUNE system using both the synchronous and asynchronous approach. The results from the implementations of these two approaches, such as the timing needed to solve the problem, number of iterations required, the running time overheads and the speed-up ratios are studied and compared later.

Using the sequential decomposition technique, shared memory is used to hold the input, the results from the first sweep and the final output component values. These values can then be accessed by different processes. Before the process iterates on its subset, it needs to read all its components first, then it releases all the values of the components for the next iteration. In the different parallel versions, different mesh sizes are evaluated, these sizes are $h^{-1}=25,49,73,97,121$, 145 and 165. In our implementations, the accuracy value is taken to be equal to $\left(5 \times 10^{-6}\right)$ and the results shown in this section (such as timing, number of iterations,...) are an average of many runs.

The First Strategy of the Parallel A.G.E. Method
In the first strategy, the problem domain (mesh of points) is decomposed into subsets each of which are assigned to a parallel path. If $P$ is the number of parallel paths and $N$ is the size of the problem,
i.e., the number of points in a given interval which is divisible by $P$, then each path works on a subset of mesh points $N_{p}=\frac{N}{P}$. This means $P$ subsets are formed with each $N_{p}$ points of the original mesh points. Each processor then computes its own subsets in two sweeps. In the first sweep, each processor evaluates its points by taking up each two successive points at a time starting from the first point and terminates after evaluating the last two points. While the second sweep is started after the first sweep has been completed. In the second sweep each processor starts its processing within its subset by evaluating the first point then each successive two points at a time and the last point is evaluated on its own. After the completion of both the first and second sweep, i.e. one cycle (iteration) is completed and a convergence test i.e. Checks to ensure that the components are obtained within the required accuracy, is carried out. As an example, given the interval shown in Figure (5.1) and by taking $P=1$, one processor will be needed (number of parallel paths equal number of processors) and starts evaluating the mesh points by taking a pair of points at a time and in the following order,

$$
\left(x_{1}, x_{2}\right),\left(x_{3}, x_{4}\right),\left(x_{5}, x_{6}\right), \ldots,\left(x_{N-1}, x_{N}\right)
$$

and this is denoted as the first sweep. A second sweep is started after the first sweep is completed and the processor will evaluate its mesh points in the second sweep in the following order,

$$
\left(x_{1}\right),\left(x_{2}, x_{3}\right),\left(x_{4}, x_{5}\right), \ldots,\left(x_{N-2}, x_{N-1}\right),\left(x_{N}\right) .
$$

So a single cycle (iteration) is terminated after evaluating all the points in the given interval in both first and second sweeps.

This strategy is implemented using both synchronous and asynchronous approaches. In the synchronous implementation approach each processor
evaluates its own subset in the manner discussed above and synchronises itself after each iteration step (i.e. after both first and second sweep). When all the processors are synchronised, the convergence test is performed by one processor, the master processor (processor 0 in the NEPTUNE system). If all the components of the mesh are obtained with the required accuracy then the procedure terminates otherwise a new iteration is needed and so on until all the components have converged.

While in the asynchronous approach the implementation strategy is by letting each processor to run asynchronously on its own subset without waiting for the other processors to complete their computations. In this case each processor iterates permanently on its subset until this subset and the other subsets which are carried out by other processors have converged. Thus, a flag is assigned to each processor, where the set of all flags is in the shared memory and can be accessed by all the processors, in order to check whether all the subsets have converged or not. At the end of each iteration (after the completion of first and second sweep), each processor checks to ensure whether its components have converged. If convergence is obtained, the processor sets its flag and tests the remaining flags to ensure that the other subsets are also converged. If not, further iteration will be required.

The Second Strategy of the Parallel A.G.E. Method
As in the first strategy, the problem in this approach is also decomposed into subsets each of which are assigned to a parallel path. If $P$ is the number of parallel paths, then each path works on a subset of mesh points $N=\frac{N}{P}$, where $N$ is the size of the problem. In the second strategy, each processor computes its own subset in two sweeps. In the
first sweep, each processor within its subset evaluates all the odd points first then followed by all the even points. In the second sweep, the evaluation is carried out in the same manner as in the first sweep, i.e., odd points are evaluated first and then followed by the even ones. As an example, given the interval shown in Figure (5.1), by taking $N=12$ and one parallel path, in the first sweep the processor starts evaluating the odd mesh points in the following order $x_{1}, x_{3}, x_{5}, x_{7}, x_{9}$ and $x_{11}$ then followed by the even points in the order of $x_{2}, x_{4}, x_{6}, x_{8}, x_{10}$ and $x_{12}$. In the second sweep, the processor first evaluates the odd points and in the order $x_{1}, x_{3}, x_{5}, x_{7}, x_{9}$ and $x_{11}$ then followed by the even points in the order of $x_{2}, x_{4}, x_{6}, x_{8}, x_{10}$ and $x_{12}$. The second strategy is implemented using both synchronous and asynchronous approaches as in the first strategy.

The above two parallel strategies were implemented on the NEPTUNE system to solve linear and non-linear two-point boundary value problems.

## Problem I

We now consider the linear problem,

$$
\begin{align*}
& U_{1}^{\prime}=U_{2}^{\prime}  \tag{5.4.1}\\
& U_{2}^{\prime}=400\left(U_{1}+\cos ^{2}(\pi x)+2 \pi^{2} \cos (2 \pi x)\right), \tag{5.4.2}
\end{align*}
$$

subject to the boundary conditions,

$$
\begin{equation*}
U_{1}(0)=U_{1}(1)=0 \tag{5.4.3}
\end{equation*}
$$

The exact solution for this problem is given by,

$$
\left.\begin{array}{l}
U_{1}(x)=\frac{e^{-20}}{1+e^{-20}} \cdot e^{20 x}+\frac{1}{1+e^{-20}} \cdot e^{-20 x}-\cos ^{2}(\pi x)  \tag{5.4.4}\\
U_{2}(x)=\frac{20 e^{-20} e^{20 x}}{1+e^{-20}}-\frac{20}{1+e^{-20}} \cdot e^{-20 x}+\pi \sin (2 \pi x)
\end{array}\right\}
$$

From (5.3.1) and (5.3.2), we have,

$$
\begin{equation*}
\left.U_{1}^{\prime \prime}=400\left(U_{1}+\cos ^{2}(\pi x)\right)+2 \pi^{2}(2 \pi x)\right) \tag{5.4.5}
\end{equation*}
$$

By following the finite difference procedure of Section (5.1), equation (5.4.5) can be approximated to obtain the linear difference equation (assuming $u=u_{1}$ ),

$$
\begin{equation*}
\frac{u_{i-1}-2 u_{i}+u_{i+1}}{h^{2}}=400\left[u_{i}+\cos ^{2}\left(\pi x_{i}\right)\right]+2 \pi^{2} \cos \left(2 \pi x_{i}\right), i=1,2, \ldots, N . \tag{5.4.6}
\end{equation*}
$$

Equation (5.4.6) can be simplified to the form,

$$
\begin{equation*}
-u_{i-1}+\left(2+400 h^{2}\right) u_{i}-u_{i+1}=-2 h^{2}\left[200 \cos ^{2}\left(\pi x_{i}\right)+\pi^{2} \cos \left(2 \pi x_{i}\right)\right], i=1,2, \ldots, N \tag{5.4.7}
\end{equation*}
$$

The boundary conditions are replaced by the values,

$$
\begin{equation*}
u_{0}=0, u_{N+1}=0, \tag{5.4.8}
\end{equation*}
$$

where $h=\frac{1}{N+1}$.
The linear system (5.4.7) can be represented in the matrix notation as,

$$
\begin{equation*}
\mathrm{A} \underline{\underline{u}}=\underline{\mathrm{b}}, \tag{5.4.9}
\end{equation*}
$$

where,


$$
\begin{align*}
& \underline{u}=\left(u_{1}, u_{2}, \ldots, u_{N-1}, u_{N}\right)^{T},  \tag{5.4.10b}\\
& \underline{g}=1+200 h^{2},
\end{align*}
$$

$$
\begin{equation*}
(5.4 .10 c) \tag{5.4.10d}
\end{equation*}
$$

and $\quad \underline{b}=\left(c_{1}, c_{2}, \ldots, c_{N-1}, c_{N}\right)^{T}$,
where,

$$
\begin{equation*}
c_{i}=-2 h^{2}\left[200 \cos ^{2}\left(\pi x_{i}\right)+\pi^{2} \cos \left(2 \pi x_{i}\right)\right], i=1,2, \ldots, N \tag{5.4.10e}
\end{equation*}
$$

We now split the matrix A into the form (5.3.1), hence from (5.3.9) $\bar{G}_{1}$ and $\bar{G}_{2}$ have the form,

with

$$
G=\left[\begin{array}{ll}
9 & -1  \tag{5.4.12}\\
-1 & g
\end{array}\right]
$$

Hence by applying the A.G.E. method, $u^{\left(k+\frac{1}{2}\right)}$ and $u^{(k+1)}$ can be determined successively by,

$$
\begin{align*}
& \underline{u}^{\left(k+\frac{1}{2}\right)}=\left(\bar{G}_{1}+r I\right)^{-1}\left[\underline{b}-\left(\bar{G}_{2}-r I\right) \underline{u}^{(k)}\right], \\
& \underline{u}^{(k+1)}=\left(\bar{G}_{2}+r I\right)^{-1}\left[\underline{b}-\left(\bar{G}_{1}-r I\right) \underline{u}^{\left(k+\frac{1}{2}\right)}\right], \tag{5.4.13}
\end{align*}
$$

where $r$ is the iteration parameter.
It is clear that $\left(\bar{G}_{1}+r I\right),\left(\bar{G}_{2}+r I\right),\left(\bar{G}_{1}-r I\right)$ and $\left(\bar{G}_{2}-r I\right)$ can be determined and $\left(\bar{G}_{1}+r I\right)^{-1},\left(\bar{G}_{2}+r I\right)^{-1}$ are easily invertible, as shown below,

[^7]where $\alpha=g+r, \beta=g-r, I$ is the identity matrix and the ( $2 \times 2$ ) submatrices of $\hat{G}_{1}, \hat{G}_{2}, \check{G}_{1}$ and $\check{G}_{2}$ have the forms,
\[

\hat{G}=\left[$$
\begin{array}{ll}
\alpha & -1  \tag{5.4.18}\\
-1 & \alpha
\end{array}
$$\right] \quad and \quad \check{G}=\left[$$
\begin{array}{ll}
\beta & -1 \\
-1 & \beta
\end{array}
$$\right]
\]

hence,

with

$$
\hat{G}^{-1}=\left[\begin{array}{ll}
\alpha \mathrm{d} & \mathrm{~d}  \tag{5.4.21}\\
\mathrm{~d} & \alpha \mathrm{~d}
\end{array}\right]=\mathrm{d}\left[\begin{array}{ll}
\alpha & 1 \\
1 & \alpha
\end{array}\right] \text {, where } \mathrm{d}=\frac{1}{\alpha^{2}-1}
$$

Hence, using equation (5.4.13) the vector $\underline{u}^{(k+1)}$ can be determined from $\underline{u}^{(k)}$ in two steps, we first determine $\underline{u}^{\left(k+\frac{1}{2}\right)}$ as follows,


Problem I was implemented in parallel on the NEPTUNE system using both strategies with synchronous and asynchronous approaches. In all these parallel implementations a different number of points within a given interval was taken, these are $h^{-1}=25,49,73,97,121,145$ and 169 . The accuracy value $(\varepsilon)$ taken to be equal to $\left(5 \times 10^{-6}\right)$ and in all these implementations the optimal iteration parameter ( $r$ ) (equation (5.3.28)) was used. The optimal iteration parameter $(x)$ is also obtained from the numerical experiments by choosing the one that gives the best running time.

The parallel synchronous strategy implementation was programmed in Program 5.1. Table (5.1) represents the results obtained from this implementation by using a number of parallel paths greater than or equal to the number of available processors using two problem sizes for equal to 24 and 48. From Table (5.1) it is clear that the optimal timing results are obtained when the number of parallel paths is equal to the number of available processors. This is due to the overheads incurred by the system to create the extra parallel paths and the waiting time for the parallel paths to be executed by an available
processor. As a consequence to this result the number of parallel paths will be taken to be equal to the number of available processors in all the other implementations.

| Size | $r$ | No.of <br> Parallel <br> Paths | No. of <br> Processors | $\begin{aligned} & \text { Time } \\ & \text { (seconds) } \end{aligned}$ | No. 0 f <br> iterations | Total no. of Parallel Paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | 0.45 | 1 | 1 | 2.10 | 8 | 34. |
|  |  | 2 | $\begin{aligned} & 1 \\ & 2 \end{aligned}$ | $\begin{aligned} & 2.12 . \\ & 1.30 \end{aligned}$ | $8$ | $\begin{gathered} 50 \\ 34,17 \end{gathered}$ |
|  |  | 3 | $\begin{aligned} & 1 \\ & 2 \\ & 3 \end{aligned}$ | $\begin{aligned} & \hline 2.15 \\ & 1.44 \\ & 0.99 \end{aligned}$ | $8$ | $\begin{gathered} 66 \\ 50,17 \\ 34,17,17 \end{gathered}$ |
|  |  | 4 | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 2.17 \\ & 1.33 \\ & 1.10 \\ & 0.82 \end{aligned}$ | $\begin{aligned} & 8 \\ & 8 \\ & 8 \\ & 8 \end{aligned}$ | $\begin{gathered} 82 \\ 50,33 \\ 50,17,17 \\ 34,17,37,17 \end{gathered}$ |
|  |  | 6 | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 2.21 \\ & 1.34 \\ & 1.00 \\ & 0.98 \end{aligned}$ | $8$ | $\begin{gathered} 114 \\ 66,49 \\ 50,33,33 \\ 50,22,24,21 \end{gathered}$ |
|  |  | 8 | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 2.25 \\ & 1.67 \\ & 1.50 \\ & 1.32 \end{aligned}$ | $\begin{aligned} & 8 \\ & 8 \\ & 8 \\ & 8 \end{aligned}$ | $\begin{gathered} 146 \\ 86,61 \\ 66,39,43 \\ 57,27,33,32 \end{gathered}$ |
|  |  | 12 | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 2.36 \\ & 1.42 \\ & 1.08 \\ & 0.90 \end{aligned}$ | $8$ | $\begin{gathered} 210 \\ 114,97 \\ 83,64,65 \\ 68,48,48,49 \end{gathered}$ |

TABLE 5.1a: The results from the synchronous strategy I using a number of parallel paths greater than the number of processors

| Size | $r$ | No. 0 f <br> Parallel <br> Paths | No. 0 f Processors | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | Total parallel Paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 48 | 0.25 | 1 | 1 | 7.85 | 15 | 62 |
|  |  | 2 | $\begin{aligned} & 1 \\ & 2 \end{aligned}$ | $\begin{aligned} & 7.90 \\ & 4.38 \end{aligned}$ | $\begin{aligned} & 15 \\ & 15 \end{aligned}$ | $\begin{gathered} 92 \\ 62,31 \end{gathered}$ |
|  |  | 3 | $\begin{aligned} & 1 \\ & 2 \\ & 3 \end{aligned}$ | $\begin{aligned} & \hline 7.94 \\ & 5.27 \\ & 3.2 \end{aligned}$ | $\begin{aligned} & 15 \\ & 15 \\ & 15 \end{aligned}$ | $\begin{gathered} 122 \\ 92,31 \\ 62,31,31 \end{gathered}$ |
|  |  | 4 | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 7.97 \\ & 4.45 \\ & 3.98 \\ & 2.51 \end{aligned}$ | $\begin{aligned} & 15 \\ & 15 \\ & 15 \\ & 15 \end{aligned}$ | $\begin{gathered} \hline 152 \\ 92,61 \\ 92,31,31 \\ 62,31,31,31 \end{gathered}$ |
|  |  | 6 | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 8.07 \\ & 4.49 \\ & 3.2 \\ & 3.13 \end{aligned}$ | $\begin{aligned} & 15 \\ & 15 \\ & 15 \\ & 15 \end{aligned}$ | $\begin{gathered} 212 \\ 122,91 \\ 92,61,61 \\ 92,43,42,38 \end{gathered}$ |
|  |  | 8 | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 8.16 \\ & 4.56 \\ & 3.47 \\ & 2.59 \end{aligned}$ | $\begin{aligned} & 15 \\ & 15 \\ & 15 \\ & 15 \end{aligned}$ | $\begin{gathered} 272 \\ 152,121 \\ 122,75,77 \\ 92,61,61,61 \end{gathered}$ |
|  |  | 12 | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 8.54 \\ & 4.64 \\ & 3.33 \\ & 2.66 \end{aligned}$ | $\begin{aligned} & 15 \\ & 15 \\ & 15 \\ & 15 \end{aligned}$ | $\begin{gathered} 392 \\ 212,181 \\ 152,121,121 \\ 122,91,91,91 \end{gathered}$ |
|  |  | 16 | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 8.54 \\ & 5.96 \\ & 4.95 \\ & 4.57 \end{aligned}$ | $\begin{aligned} & 15 \\ & 15 \\ & 15 \\ & 15 \end{aligned}$ | $\begin{gathered} \hline 512 \\ 272,241 \\ 211,152,151 \\ 152,121,121,121 \end{gathered}$ |
|  |  | 24 | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 9.06 \\ & 5.07 \\ & 3.69 \\ & 2.99 \end{aligned}$ | $\begin{aligned} & 15 \\ & 15 \\ & 15 \\ & 15 \end{aligned}$ | 752 392,361 $273,240,241$ $213,181,181,181$ |

TABLE 5.1b: The results from the synchronous strategy I using a number of parallel paths greater than the number of processors.

The results obtained from running Program 5.1 (Problem I using the first strategy with the synchronous approach) for different sizes is shown in Table 5.2. From Table 5.2 the best efficiency results (speedup) is obtained when the problem size is equal to 120.

The same strategy (Stragegy I) was implemented and programmed in Program 5.2 using the asynchronous approach by taking the number of parallel paths equal to the number of co-operating processors. Table 5.3 shows the results obtained from Program 5.2 using the same sizes that are used in the synchronized approach. Results from Table 5.3 show the best speed-up obtained when the problem size is equal to 120.

By comparing the results from Tables 5.2 and 5.3 we notice that the times for Program 5.2 are less than that of Program 5.1, i.e. evaluating the points using Strategy I with the asynchronous approach takes less time to converge than that of the synchronous approach and this is due to the synchronisation overheads needed after each iteration in the synchronous implementation. Also, from both tables it is clear that the better efficiency can be obtained by using the asynchronous approach rather than the synchronous approach. This is because the speed-up ratios of asynchronous implementation is higher than that of a synchronous one. So we can say that, using the first strategy inspite of the efficient implementation of both the synchronous and asynchronous programs, the asynchronous implementation gives better results in both the time needed for the problem to converge and on the speed-up ratios of the processors. This is due to the synchronisation overheads incurred by the system in the synchronous implementation.

Now, the second strategy was programmed and implemented on the NEPTUNE system to solve problem I using both the synchronous and

| $\begin{aligned} & \text { Mesh } \\ & \text { Size } \end{aligned}$ | $\varepsilon$ | $r$ | P | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $5 \times 10^{-6}$ | $\begin{aligned} & 0.45 \\ & 0.45 \\ & 0.45 \\ & 0.45 \end{aligned}$ | 1 2 3 4 | $\begin{aligned} & 2.10 \\ & 1.30 \\ & 0.99 \\ & 0.82 \end{aligned}$ | $\begin{aligned} & 8 \\ & 8 \\ & 8 \\ & 8 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.61539 \\ & 2.12121 \\ & 2.56098 \end{aligned}$ | $\begin{gathered} 34 \\ 34,17 \\ 34,17,17 \\ 34,17,17,17 \end{gathered}$ | $\begin{gathered} 8 \\ 8,8 \\ 8,8,8 \\ 8,8,8,8 \end{gathered}$ |
| 48 | $5 \times 10^{-6}$ | $\begin{aligned} & 0.25 \\ & 0.25 \\ & 0.25 \\ & 0.25 \end{aligned}$ | 1 <br> 2 <br> 3 <br> 4 | $\begin{aligned} & 7.85 \\ & 4.38 \\ & 3.20 \\ & 2.51 \end{aligned}$ | $\begin{aligned} & 15 \\ & 15 \\ & 15 \\ & 15 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.79224 \\ & 2.45313 \\ & 3.12749 \\ & \hline \end{aligned}$ | $\begin{gathered} 62 \\ 62,31 \\ 62,31,31 \\ 62,31,31,31 \end{gathered}$ | $\begin{gathered} 15 \\ 15,15 \\ 15,15,15 \\ 15,15,15,15 \end{gathered}$ |
| 72 | $5 \times 10^{-6}$ | $\left\{\begin{array}{l} 0.30 \\ 0.20 \\ 0.25 \\ 0.20 \end{array}\right.$ | 1 2 3 4 | $\begin{array}{r} 17.48 \\ 9.46 \\ 6.63 \\ 5.30 \\ \hline \end{array}$ | $\begin{aligned} & 22 \\ & 22 \\ & 22 \\ & 22 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.84778 \\ & 2.63650 \\ & 3.29811 \end{aligned}$ | $\begin{gathered} 90 \\ 90,45 \\ 90,45,45 \\ 90,45,45,45 \end{gathered}$ | $\begin{gathered} 22 \\ 22,22 \\ 22,22,22 \\ 22,22,22,22 \end{gathered}$ |
| 96 | $5 \times 10^{-6}$ | $\left\lvert\, \begin{aligned} & 0.15 \\ & 0.15 \\ & 0.20 \\ & 0.15 \end{aligned}\right.$ | 1 2 3 4 | $\begin{array}{r} 29.74 \\ 15.86 \\ 11.20 \\ 8.62 \\ \hline \end{array}$ | $\begin{array}{r} 28 \\ 28 \\ 28 \\ 28 \\ \hline \end{array}$ | $\begin{aligned} & 1.0 \\ & 1.87516 \\ & 2.65536 \\ & 3.45012 \\ & \hline \end{aligned}$ | $\begin{gathered} 114 \\ 114,57 \\ 114,57,57 \\ 114,57,57,57 \end{gathered}$ | $\begin{gathered} 28 \\ 28,28 \\ 28,28,28 \\ 28,28,28,28 \end{gathered}$ |
| 120 | $5 \times 10^{-6}$ | $\left\{\begin{array}{l} 0.15 \\ 0.15 \\ 0.15 \\ 0.15 \end{array}\right.$ | 1 2 3 4 | $\begin{aligned} & 46.52 \\ & 24.56 \\ & 17.18 \\ & 13.24 \end{aligned}$ | $\begin{aligned} & 35 \\ & 35 \\ & 35 \\ & 35 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.89414 \\ & 2.70780 \\ & 3.51360 \end{aligned}$ | $\begin{gathered} 142 \\ 142,71 \\ 142,71,71 \\ 142,71,71,71 \end{gathered}$ | $\begin{gathered} 35 \\ 35,35 \\ 35,35,35 \\ 35,35,35,35 \end{gathered}$ |
| 144 | $5 \times 10^{-6}$ | $\left\{\begin{array}{l} 0.15 \\ 0.15 \\ 0.15 \\ 0.15 \end{array}\right.$ | 1 2 3 4 | $\begin{aligned} & 62.05 \\ & 33.05 \\ & 23.43 \\ & 17.97 \end{aligned}$ | $\begin{aligned} & 39 \\ & 39 \\ & 39 \\ & 39 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.87746 \\ & 2.64831 \\ & 3.45298 \end{aligned}$ | $\begin{gathered} 158 \\ 158,79 \\ 158,79,79 \\ 158,79,79,79 \end{gathered}$ | $\begin{aligned} & 39 \\ & 39,39 \\ & 39,39,39 \\ & 39,39,39,39 \end{aligned}$ |
| 168 | $5 \times 10^{-6}$ | $\begin{aligned} & 0.1 \\ & 0.1 \\ & 0.1 \\ & 0.1 \end{aligned}$ | 1 2 3 4 | $\begin{aligned} & 89.25 \\ & 53.03 \\ & 37.11 \\ & 27.84 \end{aligned}$ | $\begin{aligned} & 48 \\ & 48 \\ & 48 \\ & 48 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.68301 \\ & 2.40501 \\ & 3.20582 \end{aligned}$ | $\begin{gathered} 194 \\ 194,97 \\ 194,97,97 \\ 194,97,97,97 \end{gathered}$ | $\begin{gathered} 48 \\ 48,48 \\ 48,48,48 \\ 48,48,48,48 \end{gathered}$ |

TABLE 5.2: The results from the parallel A.G.E. method using Strategy I with the synchronous approach (Problem I)

| $\begin{aligned} & \text { Mesh } \\ & \text { Size } \end{aligned}$ | $\varepsilon$ | $\mathbf{r}$ | P | $\int \begin{gathered} \text { Time } \\ (\text { seconds }) \end{gathered}$ | No. of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $5 \times 10^{-6}$ | $\begin{aligned} & 0.45 \\ & 0.45 \\ & 0.55 \\ & 0.60 \end{aligned}$ | 1 2 3 4 | $\begin{aligned} & 2.07 \\ & 1.20 \\ & 0.96 \\ & 0.77 \end{aligned}$ | $\begin{array}{r} 8 \\ 9 \\ 11 \\ 11 \end{array}$ | $\begin{gathered} 1.0 \\ 1.72500 \\ 2.15625 \\ 2.68831 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 48 | $5 \times 10^{-6}$ | $\begin{aligned} & 0.25 \\ & 0.25 \\ & 0.25 \\ & 0.25 \end{aligned}$ | 1 2 3 4 | $\begin{aligned} & 7.79 \\ & 4.22 \\ & 3.12 \\ & 2.35 \end{aligned}$ | $\begin{aligned} & 15 \\ & 16 \\ & 18 \\ & 18 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.84597 \\ 2.49680 \\ 3.31489 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 72 | $5 \times 10^{-6}$ | $\begin{aligned} & 0.25 \\ & 0.30 \\ & 0.20 \\ & 0.25 \end{aligned}$ | 1 2 3 4 | $\begin{array}{r} 17.36 \\ 9.14 \\ 6.55 \\ 5.20 \\ \hline \end{array}$ | $\begin{aligned} & 22 \\ & 23 \\ & 24 \\ & 26 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.89934 \\ 2.65038 \\ 3.33846 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 96 | $5 \times 10^{-6}$ | $\begin{aligned} & 0.15 \\ & 0.15 \\ & 0.15 \\ & 0.20 \end{aligned}$ | 1 2 3 4 | $\begin{array}{r} 29.66 \\ 15.43 \\ 11.17 \\ 8.59 \end{array}$ | $\begin{aligned} & 28 \\ & 29 \\ & 32 \\ & 34 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.92220 \\ 2.65533 \\ 3.45285 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 120 | $5 \times 10^{-6}$ | $\begin{aligned} & 0.15 \\ & 0.10 \\ & 0.15 \\ & 0.20 \end{aligned}$ | 1 2 3 4 | $\begin{aligned} & 46.36 \\ & 23.96 \\ & 17.08 \\ & 13.11 \end{aligned}$ | $\begin{aligned} & 35 \\ & 36 \\ & 38 \\ & 42 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.93489 \\ 2.71429 \\ 3.53623 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 144 | $5 \times 10^{-6}$ | $\begin{aligned} & 0.15 \\ & 0.10 \\ & 0.15 \\ & 0.10 \end{aligned}$ | 1 2 3 4 | $\begin{aligned} & 61.90 \\ & 32.95 \\ & 23.09 \\ & 17.88 \end{aligned}$ | $\begin{aligned} & 39 \\ & 42 \\ & 47 \\ & 51 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.87860 \\ 2.68081 \\ 3.46197 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 168 | $5 \times 10^{-6}$ | $\left\lvert\, \begin{aligned} & 0.10 \\ & 0.10 \\ & 0.10 \\ & 0.10 \end{aligned}\right.$ | 1 2 3 4 | $\begin{aligned} & 89.03 \\ & 52.36 \\ & 36.68 \\ & 27.39 \end{aligned}$ | $\begin{aligned} & 48 \\ & 56 \\ & 58 \\ & 63 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.70034 \\ 2.42721 \\ 3.25046 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |

TABLE 5.3: The results from the parallel A.G.E. methods using strategy I with the asynchronous approach (problem I)
asynchronous approach. In our implementation we take the number of parallel paths equal to the number of available processors. Also the accuracy value $(\varepsilon)$ is taken to be equal to $\left(5 \times 10^{-6}\right.$ ) with the optimal iteration parameter. The sizes of the problem to be solved are taken to be the same as those in the first strategy, i.e. $h^{-1}=25,49,73,97,121,145$ and 169. The results from the synchronous implementation is shown in Table 5.4 while the results from the asynchronous implementation is shown in Table 5.5. Comparing the results from these two implementations, it is noticed that the timing of asynchronous implementation using one processor is better than that of the synchronous implementation. While using more than one processor (i.e. two, three or four processors) a better timing result is obtained in the case of the synchronous implementation. It is clear from the results that in the case of the asynchronous implementation the optimal speed-up ratios obtained when the size of the problem is equal to 24 and the speed-up starts dropping when using higher problem sizes. While for the synchronous implementation a better speed-up ratio is obtained when a larger problem size is used. Also, generally we can say that in strategy II the synchronous implementation is better than an asynchronous one. These results are due to the way in which the points are evaluated in strategy II. For the asynchronous implementation, because in one iteration it first evaluates the odd points, this means it is always the old values for the even points that are used in the evaluation and vice versa. This means that an extra iteration will be needed before the recent values are used. While in the synchronous implementation, even if we evaluate the odd points first, it is guaranteed that for the next iteration new values will always be used due to the synchronisation process at the end of each iteration.

| Size | $\varepsilon$ | $r$ | P | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $5 \times 10^{-6}$ | 0.45 | 1 | 3.02 | 8 | 1.0 | 66 | 8 |
|  |  | 0.45 | 2 | 1.89 | 8 | 1.59788 | 66,33 | 8,8 |
|  |  | 0.45 | 3 | 1.48 | 8 | 2.04054 | 66,33,33 | 8,8,8 |
|  |  | 0.45 | 4 | 1.23 | 8 | 2.45529 | 66,33,33,33 | 8,8,8,8 |
| 48 | $5 \times 10^{-6}$ | 0.40 | 1 | 11.35 | 15 | 1.0 | 122 | 15 |
|  |  | 0.40 | 2 | 6.40 | 15 | 1.77344 | 122,61 | 15,15 |
|  |  | 0.40 | 3 | 4.61 | 15 | 2.46204 | 122,61,61 | 15,15,15 |
|  |  | 0.40 | 4 | 3.71 | 15 | 3.05930 | 122,61,61,61 | 15,15,15,15 |
| 72 | $5 \times 10^{-6}$ | 0.30 | 1 | 25.35 | 22 | 1.0 | 178 | 22 |
|  |  | 0.30 | 2 | 13.76 | 22 | 1.84230 | 178,89 | 22,22 |
|  |  | 0.25 | 3 | 9.82 | 22 | 2.58147 | 178,89,89 | 22,22,22 |
|  |  | 0.30 | 4 | 7.71 | 22 | 3.28794 | 178,89,89,89 | 22,22,22,22 |
| 96 | $5 \times 10^{-6}$ | 0.15 | 1 | 43.24 | 28 | 1.0 | 226 | 28 |
|  |  | 0.15 | 2 | 23.21 | 28 | 1.86299 | 226,113 | 28,28 |
|  |  | 0.15 | 3 | 16.19 | 28 | 2.67078 | 226,113,113 | 28,28,28 |
|  |  | 0.20 | 4 | 12.69 | 28 | 3.40741 | 226,113,113,113 | 28,28,28,28 |
| 120 | $5 \times 10^{-6}$ | 0.15 | 1 | 67.500 | 35 | 1.0 | 282 | 35 |
|  |  | 0.15 | 2 | 35.99 | 35 | 1.87552 | 282,141 | 35,35 |
|  |  | 0.15 | 3 | 25.10 | 35 | 2.68924 | 282,141,141 | 35,35,35 |
|  |  | 0.15 | 4 | 19.44 | 35 | 3.47222 | 282,141,141,141 | 35,35,35,35 |
| 144 | $5 \times 10^{-6}$ | 0.15 | 1 | 90.10 | 39 | 1.0 | 314 | 39 |
|  |  | 0.15 | 2 | 48.25 | 39 | 1.86736 | 314,157 | 39,39 |
|  |  | 0.15 | 3 | 33.69 | 39 | 2.67438 | 314,157,157 | 39,39,39 |
|  |  | 0.15 | 4 | 25.97 | 39 | 3.46939 | 314,157,157,157 | 39,39,39,39 |
| 168 | $5 \times 10^{-6}$ | 0.10 | 1 | 130.06 | 48 | 1.0 | 386 | 48 |
|  |  | 0.10 | 2 | 68.98 | 48 | 1.85853 | 386,193 | 48,48 |
|  |  | 0.10 | 3 | 48.82 | 48 | 2.66407 | 386,193,193 | 48,48,48 |
|  |  | 0.10 | 4 | 37.69 | 48 | 3.45078 | 386,293,193,193 | 48,48,48,48 |

TABLE 5.4: The results from the parallel A.G.E. method using strategy II with the synchronous approach (Problem I)

| Size | $\varepsilon$ | $r$ | P | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $5 \times 10^{-6}$ | $\left.\begin{aligned} & 0.45 \\ & 0.6 \\ & 0.6 \\ & 0.65 \end{aligned} \right\rvert\,$ | 1 2 3 4 | $\begin{aligned} & 2.94 \\ & 2.04 \\ & 1.45 \\ & 1.17 \end{aligned}$ | $\begin{array}{r} 8 \\ 11 \\ 11 \\ 11 \end{array}$ | $\begin{gathered} 1.0 \\ 1.44118 \\ 2.02759 \\ 2.51282 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 48 | $5 \times 10^{-6}$ | $\begin{aligned} & 0.40 \\ & 0.50 \\ & 0.45 \\ & 0.45 \end{aligned}$ | 1 2 3 4 | $\begin{array}{r} 11.13 \\ 7.52 \\ 5.38 \\ 4.42 \\ \hline \end{array}$ | $\begin{aligned} & 15 \\ & 20 \\ & 22 \\ & 24 \\ & \hline \end{aligned}$ | $\begin{array}{\|c\|} \hline 1.0 \\ 1.48005 \\ 2.06877 \\ 2.5181 \\ \hline \end{array}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \\ \hline \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 72 | $5 \times 10^{-6}$ | $\begin{aligned} & 0.25 \\ & 0.45 \\ & 0.45 \\ & 0.45 \end{aligned}$ | 1 2 3 4 | $\begin{aligned} & 24.89 \\ & 20.49 \\ & 13.93 \\ & 10.68 \\ & \hline \end{aligned}$ | $\begin{aligned} & 22 \\ & 36 \\ & 37 \\ & 39 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.21474 \\ 1.78679 \\ 2.33052 \\ \hline \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 96 | $5 \times 10^{-6}$ | $\begin{aligned} & 0.15 \\ & 0.40 \\ & 0.45 \\ & 0.45 \end{aligned}$ | 1 2 3 4 | $\begin{aligned} & 42.82 \\ & 42.65 \\ & 28.34 \\ & 24.82 \end{aligned}$ | $\begin{aligned} & 28 \\ & 56 \\ & 56 \\ & 66 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.00400 \\ 1.51094 \\ 1.72522 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 120 | $5 \times 10^{-6}$ | $\begin{aligned} & 0.15 \\ & 0.4 \\ & 0.45 \\ & 0.45 \end{aligned}$ | 1 2 3 4 | $\begin{aligned} & 67.10 \\ & 73.14 \\ & 52.29 \\ & 39.07 \end{aligned}$ | $\begin{aligned} & 35 \\ & 76 \\ & 83 \\ & 84 \end{aligned}$ | $\begin{gathered} 1.0 \\ 0.91742 \\ 1.28323 \\ 1.71743 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 144 | $5 \times 10^{-6}$ | $\begin{aligned} & 0.15 \\ & 0.35 \\ & 0.40 \\ & 0.55 \end{aligned}$ | 1 2 3 4 | $\begin{array}{r} 89.51 \\ 113.30 \\ 81.56 \\ 72.79 \\ \hline \end{array}$ | $\begin{array}{r} 39 \\ 98 \\ 107 \\ 127 \end{array}$ | $\begin{gathered} 1.0 \\ 0.79003 \\ 1.09747 \\ 1.22970 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 168 | $5 \times 10^{-6}$ | $\begin{aligned} & 0.10 \\ & 0.35 \\ & 0.40 \\ & 0.50 \end{aligned}$ | 1 2 3 4 | $\begin{aligned} & 129.110 \\ & 168.81 \\ & 124.21 \\ & 105.73 \end{aligned}$ | $\begin{array}{r} 48 \\ 125 \\ 139 \\ 155 \end{array}$ | $\begin{gathered} 1.0 \\ 0.76482 \\ 1.03945 \\ 1.22113 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |

TABLE 5.5: The results from the parallel A.G.E. method using strategy II with an asynchronous approach (Problem I)


#### Abstract

Now comparing the results obtained from solving problem I using both strategy I and II, i.e. comparing the results from Tables 5.2, 5.3, 5.4 and 5.5 , we notice that the time needed in strategy $I$ for the problem to converge is less than that of strategy II and generally the speed-up ratios using the asynchronous strategy $I$ is the best amongst the other implementations. This is mainly from the way in which the components within each strategy are evaluated. In strategy II, we first notice from its implementation that the number of computational operations are higher than that of the first strategy. Secondy, there is a possibility in the second strategy that during the evaluation of its components the old values may be used which means extra iterations will be needed. While in strategy I the most recent values of the components will be used in the evaluation process and a greater rate of convergence is achieved.

To conclude from strategy I and II we can say that in all the implementations of problem $I$ using the parallel A.G.E. method the best results are obtained when the problem is decomposed into a number of subsets, each of which is assigned to a processor where the number of processors is always equal to the number of generated parallel paths. From the implementation results we can say that strategy I will be chosen amongst the two strategies, because a better running time result was obtained. This is due to the extra amount of computational operations required in strategy II and to the way in which the evaluation of the components within each subset are carried out in each strategy, whereas in strategy I the most recent values will be used in its evaluation which is not the case in strategy II. In strategy I, because of the way in which the implementation is carried out, we can generally say that


where the timing results are concerned it does not matter whether the algorithm is synchronously or asynchronously implemented. Since we decompose the problem into almost equal subsets and assign each one to different processors, this means that the amount of work (computational time) carried out by each processor to evaluate any component is approximately the same. There are extra overheads incurred by the system besides the computational time, which degrades the parallel algorithm performance in both the synchronous and asynchronous implementations. These overheads are the generation of the parallel paths and the synchronisation at the end of each iteration cycle. These overheads may become significant, as an example, if we take the results from Tables 5.2 and 5.3 for the case when the problem size $h$ is equal to 168 . using four processors, it is clear from the asynchronous implementation that the problem converges after 63 iterations using a total of 4 parallel paths by the first processor and 2 parallel paths by each other processor. While in the case of the synchronous implementation, the same problem converges after 48 iterations using a total of 194 parallel paths by the first processor and 97 parallel paths by each of the other processors, since we need a synchronisation process after each iteration. Thus, it is clear that the overheads may effect the performance of a parallel algorithm especially in the case of the synchronous implementation. Hence for this reason we can say that the use of the asynchronous approach is better suited for the MIMD computer which confirms what is obtained from the experimental results.

The timing results from both strategy I and II are as shown diagrammatically in Figures 5.3 and 5.4 for the asynchronous approach.


FIGURE 5.3

The timing results of asynchronous strategy I and II Using parallel A.G.E. method for the sizes $24,48,72$ and 96 problem one


The timing results of asynchronous strategy | and \|I using parallel A.G.E. method for the sizes 120,144 and 168 problem one

While for the synchronous approach the results are shown in Figures 5.5 and 5.6.

For comparison reasons the parallel versions of the Jacobi, Gauss-Seidel and S.O.R. iterative methods (see Chapter 4) are implemented and programmed on the NEPTUNE system and used to solve problem I. These implementations are carried out in the same way as that of parallel A.G.E. methods using both synchronous and asynchronous approaches. Also in our implementation the problem is decomposed into a subset each of which is assigned to a parallel path where the number of parallel paths is always equal to the number of co-operating processors. Also, the accuracy value $(\varepsilon)$ is taken to be equal to $\left(5 \times 10^{-6}\right)$ and in the parallel S.O.R. method the optimal $\omega$ is obtained from the experiments by choosing the value which gives the shortest running time.

The timing results for problem I on the NEPTUNE system using the parallel asynchronous and synchronous Jacobi method are shown in Tables 5.6 and 5.7 respectively. Tables 5.8 and 5.9 represent the results for the same problem using the parallel asynchronous and synchronous Gauss-Seidel method, while the results obtained from the parallel asynchronous and synchronous S.O.R. method are shown in Tables 5.10 and 5.11 respectively.

By comparing these results with those obtained from the parallel A.G.E. method using strategy $I$ with the synchronous and asynchronous approach (Tables 5.2 and 5.3 ), it is clear that the elapsed time using the parallel A.G.E. method gives better results in all cases. This is because the number of iterations in the parallel Jacobi, Gauss-Seidel and S.O.R. methods are much higher than that of the parallel A.G.E.


FIGURE 5.5

The timing results of synchronous strategy | and II using parallel A.G.E. method for the sizes $24,48,72$ and 96 problem one


The timing results of synchronous strategy I and II using parallel A.G.E. method for The sizes 120, 144 and 168 problem one

| Size | $\varepsilon$ | P | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 4.95 \\ & 2.59 \\ & 1.75 \\ & 1.33 \end{aligned}$ | $\begin{aligned} & 40 \\ & 41 \\ & 41 \\ & 41 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.91120 \\ & 2.82857 \\ & 3.72181 \end{aligned}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 4 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 48 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{gathered} 30.12 \\ 15.46 \\ 10.21 \\ 7.8 \\ \hline \end{gathered}$ | $\begin{aligned} & 122 \\ & 124 \\ & 122 \\ & 124 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.94825 \\ & 2.95005 \\ & 3.86154 \\ & \hline \end{aligned}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 72 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 90.79 \\ & 46.12 \\ & 30.64 \\ & 23.10 \\ & \hline \end{aligned}$ | $\begin{aligned} & 243 \\ & 246 \\ & 244 \\ & 244 \\ & \hline \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.96856 \\ & 2.96312 \\ & 3.93030 \end{aligned}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 96 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{array}{r} 197.88 \\ 100.45 \\ 66.24 \\ 50.12 \end{array}$ | $\begin{aligned} & 399 \\ & 404 \\ & 398 \\ & 401 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.96994 \\ & 2.98732 \\ & 3.94813 \end{aligned}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 120 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{array}{r} 363.38 \\ 184.02 \\ 122.06 \\ 91.68 \end{array}$ | $\begin{aligned} & 585 \\ & 590 \\ & 586 \\ & 586 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.97468 \\ & 2.97706 \\ & 3.96357 \end{aligned}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 144 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 598.37 \\ & 302.89 \\ & 200.12 \\ & 151.55 \end{aligned}$ | $\begin{aligned} & 799 \\ & 808 \\ & 800 \\ & 804 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.97554 \\ & 2.99006 \\ & 3.94833 \end{aligned}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 168 | $5 \times 10^{-6}$ | $\begin{aligned} & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 901.13 \\ & 459.05 \\ & 302.31 \\ & 231.45 \end{aligned}$ | $\begin{aligned} & 1041 \\ & 1053 \\ & 1040 \\ & 1043 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.96303 \\ & 2.98081 \\ & 3.89341 \end{aligned}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |

TABLE 5.6: The results from the parallel Jacobi iterative method using the asynchronous approach (Problem I)

| Size | $\varepsilon$ | P | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | Speed-up | Total no. of parallel paths. | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 5.05 \\ & 3.14 \\ & 2.37 \\ & 1.99 \\ & \hline \end{aligned}$ | $\begin{aligned} & 40 \\ & 40 \\ & 40 \\ & 40 \\ & \hline \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.60828 \\ 2.13080 \\ 2.53769 \\ \hline \end{gathered}$ | $\begin{gathered} 82 \\ 82,41 \\ 82,41,41 \\ 82,41,41,41 \end{gathered}$ | $\begin{gathered} 40 \\ 40,40 \\ 40,40,40 \\ 40,40,40,40 \end{gathered}$ |
| 48 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 30.47 \\ & 17.19 \\ & 12.34 \\ & 10.01 \end{aligned}$ | $\begin{aligned} & 122 \\ & 122 \\ & 122 \\ & 122 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.77254 \\ & 2.46921 \\ & 3.04396 \end{aligned}$ | $\begin{gathered} 246 \\ 246,123 \\ 246,123,123 \\ 246,123,123,123 \end{gathered}$ | $\begin{gathered} 122 \\ 122,122 \\ 122,122,122 \\ 22,122,122,122 \end{gathered}$ |
| 72 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 91.49 \\ & 49.83 \\ & 35.07 \\ & 27.78 \\ & \hline \end{aligned}$ | $\begin{aligned} & 243 \\ & 243 \\ & 243 \\ & 243 \\ & \hline \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.83604 \\ & 2.60878 \\ & 3.29338 \\ & \hline \end{aligned}$ | $\begin{gathered} 488 \\ 488,244 \\ 488,244,244 \\ 488,244244244 \end{gathered}$ | $\begin{gathered} 243 \\ 243,243 \\ 243,243,243 \\ 243,243,243,243 \\ \hline \end{gathered}$ |
| 96 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{gathered} 198.89 \\ 106.88 \\ 74.43 \\ 58.5 \\ \hline \end{gathered}$ | $\begin{aligned} & 399 \\ & 399 \\ & 399 \\ & 399 \\ & \hline \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.86087 \\ 2.67218 \\ 3.39983 \\ \hline \end{gathered}$ | $\begin{gathered} 800 \\ 800,400 \\ 800,400,400 \\ 300400400400 \\ \hline \end{gathered}$ | $\begin{gathered} 399 \\ 399,399 \\ 399,399,399 \\ 399,399,399,399 \\ \hline \end{gathered}$ |
| 120 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 365.12 \\ & 193.98 \\ & 134.60 \\ & 105.33 \end{aligned}$ | $\begin{aligned} & 585 \\ & 585 \\ & 585 \\ & 585 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.88226 \\ & 2.71263 \\ & 3.46644 \end{aligned}$ | 1172 1172,586 $172,586,586$ $172,586,586,586$ | $\begin{gathered} 585 \\ 585,585 \\ 585,585,585 \\ 585,585,585,585 \end{gathered}$ |
| 144 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \\ & \hline \end{aligned}$ | $\begin{aligned} & 599.00 \\ & 317.31 \\ & 219.92 \\ & 171.14 \\ & \hline \end{aligned}$ | $\begin{aligned} & 799 \\ & 799 \\ & 799 \\ & 799 \\ & \hline \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.88774 \\ & 2.72371 \\ & 3.50006 \end{aligned}$ | $\begin{array}{\|l\|} 1600 \\ 1600,800 \\ 1600,800,800 \\ 1600800800800 \end{array}$ | $\begin{gathered} 799 \\ 799,799 \\ 799,799,799 \\ 799,799,799,799 \end{gathered}$ |
| 168 | $5 \times 10^{-6}$ | 1 2 3 4 | $\begin{aligned} & 903.50 \\ & 480.63 \\ & 332.66 \\ & 259.10 \end{aligned}$ | $\begin{aligned} & 1041 \\ & 1041 \\ & 1041 \\ & 1041 \end{aligned}$ | $\begin{aligned} & 1.0 \\ & 1.87982 \\ & 2.71599 \\ & 3.48707 \end{aligned}$ | $\|$2084 <br> 2084,1042 <br> $2084,1042,1042$ <br> 2084,1042, <br> 1042,1042 | 1041 1041,1041 $1041,1041,1041$ $1041,1041,1041$ 1041 |

TABLE 5.7: The results from the parallel Jacobi iterative method using the synchronous approach (Problem I)

| Size | $\varepsilon$ | P | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 2.98 \\ & 1.59 \\ & 1.04 \\ & 0.79 \\ & \hline \end{aligned}$ | $\begin{aligned} & 24 \\ & 25 \\ & 24 \\ & 24 \\ & \hline \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.87421 \\ 2.86539 \\ 3.77215 \\ \hline \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \\ \hline \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 48 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \\ & \hline \end{aligned}$ | $\begin{gathered} 16.79 \\ 8.60 \\ 5.7 \\ 4.33 \\ \hline \end{gathered}$ | $\begin{aligned} & 68 \\ & 69 \\ & 68 \\ & 68 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.95236 \\ 2.94561 \\ 3.87760 \\ \hline \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \\ \hline \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 72 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 49.98 \\ & 25.13 \\ & 16.72 \\ & 12.73 \\ & \hline \end{aligned}$ | $\begin{aligned} & 134 \\ & 134 \\ & 133 \\ & 135 \\ & \hline \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.98886 \\ 2.98924 \\ 3.92616 \\ \hline \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \\ \hline \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 96 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{array}{r} 108.40 \\ 54.76 \\ 36.24 \\ 27.37 \end{array}$ | $\begin{aligned} & 219 \\ & 220 \\ & 218 \\ & 220 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.97955 \\ 2.99117 \\ 3.96054 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \\ \hline \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 120 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{array}{r} 199.21 \\ 100.51 \\ 66.67 \\ 50.22 \end{array}$ | $\begin{aligned} & 321 \\ & 322 \\ & 320 \\ & 322 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.98199 \\ 2.98800 \\ 3.96675 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 144 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \\ & \hline \end{aligned}$ | $\begin{array}{r} 327.46 \\ 165.15 \\ 109.76 \\ 82.56 \\ \hline \end{array}$ | $\begin{aligned} & 439 \\ & 441 \\ & 439 \\ & 440 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.98280 \\ 2.98342 \\ 3.96633 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 168 | $5 \times 10^{-6}$ | $\begin{array}{\|l} 1 \\ 2 \\ 3 \\ 4 \end{array}$ | $\begin{aligned} & 492.21 \\ & 250.71 \\ & 166.20 \\ & 125.00 \end{aligned}$ | $\begin{aligned} & 573 \\ & 576 \\ & 572 \\ & 573 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.96326 \\ 2.96155 \\ 3.93768 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |

TABLE 5.8: The results from the parallel Gauss-Seidel iterative method using the asynchronous approach (Problem I)

| Size | $\boldsymbol{\varepsilon}$ | P | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 3.03 \\ & 1.91 \\ & 1.43 \\ & 1.21 \\ & \hline \end{aligned}$ | $\begin{aligned} & 24 \\ & 24 \\ & 24 \\ & 24 \\ & \hline \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.58639 \\ 2.11888 \\ 2.50413 \end{gathered}$ | $\begin{gathered} 50 \\ 50,25 \\ 50,25,25 \\ 50,25,25,25 \end{gathered}$ | $\begin{gathered} 24 \\ 24,24 \\ 24,24,24 \\ 24,24,24,24 \end{gathered}$ |
| 48 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{array}{r} 16.94 \\ 9.75 \\ 6.86 \\ 5.56 \\ \hline \end{array}$ | $\begin{aligned} & 68 \\ & 68 \\ & 68 \\ & 68 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.73744 \\ 2.46939 \\ 3.04676 \\ \hline \end{gathered}$ | $\begin{array}{\|c\|} 138 \\ 138,69 \\ 138,69,69 \\ 138,69,69,69 \end{array}$ | $\begin{gathered} 68 \\ 68,68 \\ 68,68,68 \\ 68,68,68,68 \\ \hline \end{gathered}$ |
| 72 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 50.28 \\ & 27.47 \\ & 19.36 \\ & 15.29 \\ & \hline \end{aligned}$ | $\begin{aligned} & 134 \\ & 134 \\ & 134 \\ & 134 \\ & \hline \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.83036 \\ 2.59711 \\ 3.28842 \\ \hline \end{gathered}$ | $\begin{gathered} 270 \\ 270,135 \\ 270,135,135 \\ 270,135,135,135 \end{gathered}$ | $\begin{gathered} 134 \\ 134,134 \\ 134,134,134 \\ 134,134,134,134 \\ \hline \end{gathered}$ |
| 96 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{array}{r} 108.93 \\ 58.47 \\ 40.62 \\ 32.02 \\ \hline \end{array}$ | $\begin{aligned} & 219 \\ & 219 \\ & 219 \\ & 219 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.86301 \\ 2.68168 \\ 3.40194 \\ \hline \end{gathered}$ | 440 440,220 $440,220,220$ $440,220,220,220$ | $\begin{gathered} 219 \\ 219,219 \\ 219,219,219 \\ 219,219,219,219 \\ \hline \end{gathered}$ |
| 120 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{array}{r} 199.88 \\ 106.22 \\ 73.69 \\ 57.57 \end{array}$ | $\begin{aligned} & 321 \\ & 321 \\ & 321 \\ & 321 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.88176 \\ 2.71244 \\ 3.47195 \end{gathered}$ | $\begin{gathered} 644 \\ 644,322 \\ 644,322,322 \\ 644,322,322,322 \end{gathered}$ | $\begin{gathered} 321 \\ 321,321 \\ 321,321,321 \\ 321,321,321,321 \end{gathered}$ |
| 144 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{array}{r} 328.50 \\ 173.73 \\ 119.93 \\ 93.30 \end{array}$ | $\begin{aligned} & 439 \\ & 439 \\ & 439 \\ & 439 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.89087 \\ 2.73910 \\ 3.52090 \end{gathered}$ | $\left.\begin{gathered} 880 \\ 880,440 \\ 880,440,440 \\ 880,440 A 40 A 40 \end{gathered} \right\rvert\,$ | $\begin{gathered} 439 \\ 439,439 \\ 439,439,439 \\ 439,439,439,439 \end{gathered}$ |
| 168 | $5 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\left\{\begin{array}{l} 494.77 \\ 262.34 \\ 181.76 \\ 141.66 \end{array}\right.$ | $\begin{aligned} & 573 \\ & 573 \\ & 573 \\ & 573 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.88599 \\ 2.72211 \\ 3.49266 \end{gathered}$ | 1148 1148,574 $1148,574,574$ $1148,574,574$ 574 $\|$ | $\begin{gathered} 573 \\ 573,573 \\ 573,573,573 \\ 573,573,573,573 \end{gathered}$ |

TABLE 5.9: The results from the pariallel Gauss-Seidel iterative method using the synchronous approach (Problem I)

| Size | $\varepsilon$ | $\omega$ | $P$ | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No.of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $5 \times 10^{-6}$ | $\begin{aligned} & 1.20 \\ & 1.20 \\ & 1.20 \\ & 1.20 \end{aligned}$ | 1 2 3 4 | $\begin{aligned} & 2.60 \\ & 1.58 \\ & 1.09 \\ & 0.83 \end{aligned}$ | $\begin{aligned} & 17 \\ & 20 \\ & 20 \\ & 20 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.64557 \\ 2.38532 \\ 3.13253 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 48 | $5 \times 10^{-6}$ | $\begin{aligned} & 1.45 \\ & 1.45 \\ & 1.40 \\ & 1.45 \\ & \hline \end{aligned}$ | 1 2 3 4 | $\begin{aligned} & 9.78 \\ & 5.29 \\ & 3.86 \\ & 2.92 \end{aligned}$ | $\begin{aligned} & 32 \\ & 34 \\ & 37 \\ & 37 \\ & \hline \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.84877 \\ 2.53368 \\ 3.34932 \\ \hline \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \\ \hline \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 72 | $5 \times 10^{-6}$ | $\left\|\begin{array}{l} 1.55 \\ 1.60 \\ 1.60 \\ 1.55 \end{array}\right\|$ | 1 2 3 4 | $\begin{array}{r} 21.24 \\ 11.75 \\ 8.54 \\ 6.40 \end{array}$ | $\begin{aligned} & 46 \\ & 50 \\ & 53 \\ & 55 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.80766 \\ 2.48712 \\ 3.31875 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \\ \hline \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 96 | $5 \times 10^{-6}$ | $\left(\begin{array}{l} 1.65 \\ 1.65 \\ 1.65 \\ 1.65 \end{array}\right.$ | 1 2 3 4 | $\begin{aligned} & 36.87 \\ & 20.50 \\ & 14.07 \\ & 11.35 \end{aligned}$ | $\begin{aligned} & 60 \\ & 67 \\ & 68 \\ & 73 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.79854 \\ 2.62047 \\ 3.24846 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 120 | $5 \times 10^{-6}$ | $\left[\begin{array}{l} 1.70 \\ 1.70 \\ 1.70 \\ 1.70 \end{array}\right.$ | 1 2 3 4 | $\begin{aligned} & 57.06 \\ & 31.25 \\ & 22.30 \\ & 17.65 \\ & \hline \end{aligned}$ | $\begin{aligned} & 74 \\ & 80 \\ & 85 \\ & 92 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.82592 \\ 2.55874 \\ 3.23286 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 144 | $5 \times 10^{-6}$ | $\left(\begin{array}{l} 1.75 \\ 1.75 \\ 1.75 \\ 1.75 \end{array}\right.$ | 1 2 3 4 | $\begin{aligned} & 81.69 \\ & 43.97 \\ & 30.10 \\ & 24.43 \end{aligned}$ | $\begin{array}{r} 88 \\ 94 \\ 100 \\ 104 \end{array}$ | $\begin{gathered} 1.0 \\ 1.85786 \\ 2.71395 \\ 3.37264 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 168 | $5 \times 10^{-6}$ | $\left\|\begin{array}{l} 1.75 \\ 1.80 \\ 1.80 \\ 1.75 \end{array}\right\|$ | 1 2 3 4 | $\begin{array}{r} 112.69 \\ 61.00 \\ 41.80 \\ 33.88 \end{array}$ | $\begin{aligned} & 105 \\ & 112 \\ & 114 \\ & 126 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.84738 \\ 2.69593 \\ 3.32615 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |

TABLE 5.10: The results from the parallel S.O.R. method using the asynchronous approach (Problem I)

| Size | $\varepsilon$ | $\omega$ | P | $\left\|\begin{array}{c} \text { Time } \\ \text { (seconds) } \end{array}\right\|$ | No. of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $5 \times 10^{-6}$ | $\left\|\begin{array}{l} 1.21 \\ 1.23 \\ 1.23 \\ 1.21 \end{array}\right\|$ | 1 2 3 | $\begin{aligned} & 2.62 \\ & 1.69 \\ & 1.40 \\ & 1.16 \\ & \hline \end{aligned}$ | $\begin{aligned} & 17 \\ & 18 \\ & 20 \\ & 20 \\ & \hline \end{aligned}$ | $\begin{array}{\|c} 1.0 \\ 1.55029 \\ 1.87143 \\ 2.25862 \end{array}$ | $\begin{gathered} 36 \\ 38,19 \\ 42,21,21 \\ 42,21,21,21 \end{gathered}$ | $\begin{gathered} 17 \\ 18,18 \\ 20,20,20 \\ 20,20,20,20 \\ \hline \end{gathered}$ |
| 48 | $5 \times 10^{-6}$ | $\left\|\begin{array}{l} 1.44 \\ 1.45 \\ 1.40 \\ 1.44 \end{array}\right\|$ | 2 3 4 | $\begin{aligned} & 9.84 \\ & 5.81 \\ & 4.49 \\ & 3.69 \\ & \hline \end{aligned}$ | $\begin{array}{r} 32 \\ 34 \\ 37 \\ 38 \\ \hline \end{array}$ | $\begin{array}{\|c} 1.0 \\ 1.69363 \\ 2.19154 \\ 2.66667 \\ \hline \end{array}$ | $\begin{gathered} 64 \\ 70,35 \\ 76,38,38 \\ 78,39,39,39 \\ \hline \end{gathered}$ | $\begin{gathered} 32 \\ 34,34 \\ 37,37,37 \\ 38,38,38,38 \\ \hline \end{gathered}$ |
| 72 | $5 \times 10^{-6}$ | $\begin{aligned} & 1.56 \\ & 1.57 \\ & 1.55 \\ & 1.56 \end{aligned}$ | 2 <br> 3 <br> 4 | $\begin{array}{r} 21.27 \\ 12.20 \\ 9.26 \\ 7.72 \end{array}$ | $\begin{aligned} & 46 \\ & 49 \\ & 53 \\ & 56 \\ & \hline \end{aligned}$ | $\begin{array}{\|c} 1.0 \\ 1.74344 \\ 2.29698 \\ 2.75518 \\ \hline \end{array}$ | $\begin{array}{\|c\|} \hline 94 \\ 100,50 \\ 108,54,54 \\ 114,57,57,57 \\ \hline \end{array}$ | $\begin{gathered} 46 \\ 49,49 \\ 53,53,53 \\ 56,56,56,56 \\ \hline \end{gathered}$ |
| 96 | $5 \times 10^{-6}$ | $\left\|\begin{array}{l} 1.64 \\ 1.66 \\ 1.63 \\ 1.63 \end{array}\right\|$ | 1 2 3 4 4 | $\begin{aligned} & 36.99 \\ & 20.88 \\ & 15.70 \\ & 12.98 \\ & \hline \end{aligned}$ | $\begin{array}{r} 60 \\ 64 \\ 69 \\ 73 \\ \hline \end{array}$ | $\begin{array}{\|c} 1.0 \\ 1.77155 \\ 2.35605 \\ 2.84977 \\ \hline \end{array}$ | $\begin{array}{\|c\|} 122 \\ 130,65 \\ 140,70,70 \\ 148,74,74,74 \\ \hline \end{array}$ | $\begin{gathered} 60 \\ 64,64 \\ 69,69,69 \\ 73,73,73,73 \\ \hline \end{gathered}$ |
| 120 | $5 \times 10^{-6}$ | $\left\lvert\, \begin{aligned} & 1.69 \\ & 1.71 \\ & 1.70 \\ & 1.69 \end{aligned}\right.$ | 1 2 3 4 | $\begin{aligned} & 57.22 \\ & 32.33 \\ & 23.95 \\ & 19.66 \\ & \hline \end{aligned}$ | $\begin{array}{r} 74 \\ 79 \\ 85 \\ 90 \\ \hline \end{array}$ | $\begin{array}{\|c} 1.0 \\ 1.76987 \\ 2.38914 \\ 2.91048 \\ \hline \end{array}$ | 150 160,80 $172,86,86$ $182,91,91,91$ | $\begin{gathered} 74 \\ 79,79 \\ 85,85,85 \\ 90,90,90,90 \\ \hline \end{gathered}$ |
| 144 | $5 \times 10^{-6}$ | $\begin{array}{\|l\|} 1.75 \\ 1.75 \\ 1.73 \\ 1.73 \\ \hline \end{array}$ | 2 3 4 | $\begin{aligned} & 81.68 \\ & 45.86 \\ & 33.75 \\ & 27.74 \\ & \hline \end{aligned}$ | $\begin{array}{r} 88 \\ 94 \\ 101 \\ 107 \\ \hline \end{array}$ | $\begin{array}{\|c} 1.0 \\ 1.78107 \\ 2.42015 \\ 2.94449 \\ \hline \end{array}$ | 178 190,95 $204,102,102$ $216,108,108,108$ | $\begin{aligned} & 88 \\ & 94,94 \\ & 101,101,101 \\ & 107,107,107,107 \\ & \hline \end{aligned}$ |
| 168 | $5 \times 10^{-6}$ | $\left\|\begin{array}{l} 1.78 \\ 1.78 \\ 1.77 \\ 1.77 \end{array}\right\|$ | 1 2 3 4 | $\begin{array}{r} 108.74 \\ 60.87 \\ 43.70 \\ 37.36 \end{array}$ | $\begin{aligned} & 101 \\ & 108 \\ & 113 \\ & 125 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.78643 \\ 2.48833 \\ 2.91060 \end{gathered}$ | $\begin{gathered} 204 \\ 218,109 \\ 228,114,114 \\ 252,126,126,126 \end{gathered}$ | $\begin{aligned} & 101 \\ & 108,108 \\ & 113,113,113 \\ & 125,125,125,125 \end{aligned}$ |

TABLE 5.11: The results from the parallel S.O.R. method using the synchronous approach (Problem I)
method, which means more total computational operations are required. For this reason the parallel A.G.E. method is chosen to be the best amongst all the other parallel methods.

As an example, Figures 5.7 and 5.8 show the run time results using parallel A.G.E., Jacobi, Gauss-Seidel and S.O.R. methods when the size of the problem is equal to 168 using synchronous and asynchronous approaches respectively.

Now to calculate the number of arithmetic operations that are required to solve problem I by using the parallel A.G.E. method, we concentrate on strategy I using the asynchronous approach because it gives the best results amongst the other parallel implemented methods. We first calculate the number of operations required to determine the $\left(k+\frac{1}{2}\right)^{\text {th }}$ iterate of the A.G.E. method. From system (5.4.22), if we assume that,

$$
r_{1}=c_{1}-\beta u_{1}^{(k)} \quad \text { and } \quad r_{2}=c_{2}-\beta u_{2}^{(k)}+u_{3}^{(k)}
$$

then,

$$
u_{1}^{\left(k+\frac{1}{2}\right)}=d\left(\alpha r_{1}+r_{2}\right) \quad \text { and } \quad u_{2}^{\left(k+\frac{1}{2}\right)}=d\left(r_{1}+\alpha r_{2}\right)
$$

and for $i=3,2,(N-3)$

$$
u_{i}^{\left(k+\frac{1}{2}\right)}=d\left(\alpha r_{1}^{\prime}+r_{2}^{\prime}\right) \text { and } u_{i+1}^{\left(k+\frac{1}{2}\right)}=d\left(r_{1}^{\prime}+r_{2}^{\prime}\right)
$$

where $\quad r_{1}^{\prime}=c_{i}+u_{i-1}^{(k)}-\beta u_{i}^{(k)}$ and $r_{2}^{\prime}=c_{i+1}-\beta u_{i+1}^{(k)}+u_{i+2}^{(k)}$, and finally,

$$
u_{N-1}^{\left(k+\frac{1}{2}\right)}=d\left(\alpha r_{1}^{\prime \prime}+r_{2}^{\prime \prime}\right) \text { and } u_{N}^{\left(k+\frac{1}{2}\right)}=d\left(r_{1}^{\prime \prime}+r_{2}^{\prime \prime}\right) \text {. }
$$

where $\quad r_{1}^{\prime \prime}=c_{N-1}+u_{N-2}^{(k)}-\beta u_{N-1}^{(k)}$ and $r_{2}^{\prime \prime}=c_{N}-\beta u_{N}^{(k)}$.
The values of $c_{i}$, for $i=1,2, \ldots, N$ are given in equation (5.4.10e), $d$ is given in equation (5.4.21), $\alpha$ and $\beta$ are given in (5.4.17) and $N$ is divisible by 2. While the number of operations required to determine


FIGURE 5.7

The timing results of parallel Jacobi, Gauss-Siedel, S.O.R. and A.G.E. Methods using asynchronous approch for problem size equal to 168


FIGURE 5.8

The Timing Results Of Parallel Jacobi, Gauss-Siedel, S.O.R. and A.G.E Methods Using Synchronous Approch For Problem Size Equal To 168
the $(k+1)^{\text {th }}$ iterate of the A.G.E. method is obtained from system (5.4.23) if we assume that,

$$
r_{1}=c_{1}-\beta u_{1}^{\left(k+\frac{1}{2}\right)}+u_{2}^{\left(k+\frac{1}{2}\right)}
$$

then

$$
u_{1}^{(k+1)}=\frac{1}{\alpha}\left(r_{1}\right)
$$

and for $i=2,2,(N-2)$

$$
u_{i}^{(k+1)}=d\left(\alpha r_{1}^{\prime}+r_{2}^{\prime}\right) \text { and } u_{i+1}^{(k+1)}=d\left(r_{1}^{\prime}+\alpha r_{2}^{\prime}\right)
$$

where,

$$
r_{1}^{\prime}=c_{i}+u_{i-1}^{\left(k+\frac{1}{2}\right)}-\beta u_{i}^{\left(k+\frac{1}{2}\right)} \text { and } r_{2}^{\prime}=c_{i+1}-\beta u_{i+1}^{\left(k+\frac{1}{2}\right)}+u_{i+2}^{\left(k+\frac{1}{2}\right)}
$$

and finally,

$$
u_{N}^{(k+1)}=\frac{1}{\alpha}\left(r_{1}^{\prime \prime}\right)
$$

where $\quad r_{1}^{\prime \prime}=c_{N}+u_{N-1}^{\left(k+\frac{1}{2}\right)}-\beta_{N}^{\left(k+\frac{1}{2}\right)}$,
where the values of $c_{i}$ for $i=1,2, \ldots, N$ are given in equation (5.4.10e), $d$ is given in equation (5.4.21), $\alpha$ and $\beta$ are given in (5.4.17) and $N$ is divisible by 2.

Thus, the computational complexity for the sequential algorithm can be easily calculated by taking the number of operations required in each sweep. Therefore, for the first sweep the number of operations required for each point per iteration in the mesh is equal to (by denoting the multiplication and the addition by $M$ and $A$ respectively),
for the first and the last two points $=(4 M+4 A)$,
for the other points in the mesh $=(4 M+5 A)$.
For the second sweep the number of operations required for each point in the mesh per iteration is equal to,

| for the first and last points | $=(2 M+2 A)$, |
| :--- | :--- |
| for the other points in the mesh | $=(4 M+5 A)$, |

presuming that $d$ is evaluated and stored at the beginning of the program.

So, generally speaking, for each sweep we can estimate the number of operations for each point in the mesh per iteration is equal to $(4 M+4 A)$, therefore for the two sweeps the total number of operations for each point in each iteration is equal to ( $8 \mathrm{M}+8 \mathrm{~A}$ ) and by letting d be evaluated and stored at the beginning of the program.

Now for N mesh points and P parallel paths each processor will evaluate $\left(\frac{N}{P}\right)$ points with total computational complexity is equal to $T=\left[(8 M+8 A) \cdot \frac{N}{P}\right]$ operations per iteration. Besides the computational time T there are some delay times due to the overheads incurred by the system which may degrade the algorithm's performance. These overheads are the generation of the parallel paths and the synchronisation at the end of each iteration cycle.

From Chapter 4 it can be seen that the number of arithmetical operations required for the solution of problem I using the Jacobi or Gauss-Seidel iterative methods is equal to [l multiplication (M) + 3 additions (A)], while using the S.O.R. iterative method needs $[2(M)+5(A)]$. The total number of arithmetic operations required for the solution of problem I using the parallel A.G.E., Jacobi, Gauss-Seidel and S.O.R. methods using a mesh size $\mathrm{h}^{-1}=169$ are shown in Table 5.12. These results are obtained from the number of arithmetic operations required by each method and the results obtained from Tables $5.2,5.3,5.6,5.7,5.8$, 5.9, 5.10 and 5.11. In Table $5.12 \mathrm{~K}=\mathrm{N} / \mathrm{P}$, where N represents the number of points in the mesh to be solved and $P$ the number of parallel paths generated and in our case equal to the number of cooperating processors. By comparing the results obtained from Table 5.12 and the experimental results, it is clear that all the results agree apart from the case of the asynchronous S.O.R. method and synchronous S.O.R. method when using
only one or two processors. This is due to the extra overheads which are not added to the results shown in Table 5.12. These overheads are the generations of the parallel paths and the synchronization process needed after each iteration cycle, which is greater in the case of S.O.R. method due to the increased iterations required as shown in Tables 5.10 and 5.11.

| Size <br> (N) | P | A.G.E. |  |  |  | Jacobi |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | Synchronous |  | Asynchronous |  | Synchronous |  | Asynchronous |  |
|  |  | M | A | M | A | M | A | M | A |
| 168 | 1 | 384K | 384K | 384 K | 384K | 1041K | 3123K | 1041K | 3123K |
|  | 2 | 384K | 384K | 448K | 448K | 1041K | 3123K | 1053 K | 3159K |
|  | 3 | 384K | 384 K | 464K | 464K | 1041K | 3123K | 1040K | 3120K |
|  | 4 | 384 K | 384 K | 504K | 504K | 1041K | 3123K | 1043K | 3129K |

${ }^{*} K=N / P$
TABLE 5.12(a)

| Size <br> (N) | P | Gauss-Seidel |  |  |  | S.O.R. |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | Synchronous |  | Asynchronous |  | Synchronous |  | Asynchronous |  |
|  |  | M | A | M | A | M | A | M | A |
| 168 | 1 | 573K | 1719K | 573K | 1719K | 202K | 505K | 210K | 525K |
|  | 2 | 573K | 1719K | 576K | 1728K | 216K | 540K | 224 K | 560K |
|  | 3 | 573K | 1719K | 572K | 1716K | 226K | 565K | 228K | 570 K |
|  | 4 | 573K | 1719K | 573 K | 1719K | 250K | 625K | 252K | 630 K |

*K=N/P
TABLE 5.12(b)

Now for a further performance analysis of the parallel A.G.E. method when used to solve Problem $I$, we follow the same steps used in Chapter 4 using the system commands XPFCLS, XPFCLN and XPFCL. Both the
information obtained from Figure 4.15 (the resource timings of the NEPTUNE system) and the experimental results in Tables $5.2,5.3,5.4$ and 5.5 are also used in the analysis.

Actually Table 5.13 gives the mean rate of access to the shared data and parallel path scheduling. It gives estimates of the potential speed-up for using P processors, where $N$ represents the number of points in the mesh to be solved. On the other hand, Table 5.14 illustrates the results obtained when the algorithms were run on the NEPTUNE system. The parallel control access overheads (PCO) are taken for the case $P=1$ and $N$ is taken to be that which gives the best speed-up ratios (i.e. $N=48$ in the case of using the asynchronous method strategy II and $N=120$ for the other methods).

Now by examining the results in Tabels 5.13 and 5.14 , we notice that the best results are obtained when using strategy $I$ of the parallel A.G.E. method with the asynchronous approach, and we see that a linear speed-up has been achieved and as many as $N$ (the number of mesh points) processors can be employed as an upper limit. The strategy I parallel A.G.E. method with the asynchronous approach has made 17 accesses to the shared data per 65 floating point operators (flops). From the information in Table 4.15, the static shared data access overheads in this algorithm is equal to,

$$
\frac{1}{\left(\frac{65}{17}\right)} * \frac{0.75}{720} * 100=0.0278
$$

Also this method from its program made 1 access per ( 16 * $_{\mathrm{N}}^{\mathrm{p}}$ ) flops, where $N_{p}$ is equal to the number of points in each process. This results in a parallel path access loss equal to $-0.087 \%$ since the parallel path mechanism requires $\sim 1200 \mu s$. This loss is almost the same as the parallel path access loss of Table 5.14.

From Figures in Tables 5.13 and 5.14 we conclude that the best results are obtained when using the strategy I of the parallel A.G.E. method with the asynchronous approach and the predicted results obtained for the shared data and the parallel path are in almost agreement with the experimental results obtained from the NEPTUNE system.

| Program | Processors (P) |  | Shared Data |  | Parallel Path |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | No. | Speed-up | Access Rate | Overhead <br> Amount | Access Rate | Overhead Amount |
| ```Parallel A.G.E. strategy I (synchronous)``` | $P \leqslant N$ | O(P) | 17:65 <br> flops | $0.027 \%$ | $\begin{aligned} & 1:\left(16 * N_{p}\right) \\ & \text { flops } \end{aligned}$ | $0.087 \%$ |
| ```Parallel A.G.E. strategy I (asynchronous)``` | $P_{\leqslant} \leqslant$ | O(P) | 17:65 <br> flops | 0.027\% | $\begin{aligned} & 1:\left(16 * N_{p}\right) \\ & \text { flops } \end{aligned}$ | 0.087\% |
| Parallel A.G.E. strategy II (synchronous) | $\mathrm{P} \leqslant \mathrm{N}$ | O(P) | 25:72 <br> flops | 0.036\% | $\begin{aligned} & \text { I: }\left(16 * N_{p}\right) \\ & \text { flops } \end{aligned}$ | 0.087\% |
| ```Parallel A.G.E. strategy II (asynchronous)``` | $P_{\S} N$ | $O(P)$ | $25: 72$ <br> flops | 0.036\% | $\begin{aligned} & 1:\left(16 * N_{p}\right) \\ & \text { flops } \end{aligned}$ | 0.087\% |

TABLE 5.13: Resource demands of the parallel A.G.E. algorithms

| Program | Speed-up <br> Shared Data |  |  | Parallel <br> Overhead <br> (SDO) | Overhead <br> (PCO) |
| :---: | :---: | :---: | :---: | :---: | :---: |
| Parallel A.G.E. <br> strategY I <br> (synchronous) | 1.89414 | 2.70780 | 3.51360 | $0.043 \%$ | $0.172 \%$ |
| Parallel A.G.E. <br> strategy I <br> (asynchronous) | 1.93489 | 2.71429 | 3.53623 | $0.022 \%$ | $0.086 \%$ |
| Parallel A.G.E. <br> strategy II <br> (synchronous) | 1.87552 | 2.68924 | 3.47222 | $0.074 \%$ | $0.089 \%$ |
| Parallel A.G.E. <br> strategy II <br> (asynchronous) | 1.48005 | 2.06877 | 2.51810 | $0.090 \%$ | $0.090 \%$ |

TABLE 5.14: Performance measurements of parallel A.G.E. methods on the NEPTUNE system

## Problem II

We now consider the nonlinear problem,

$$
\begin{equation*}
u^{\prime \prime}=\frac{1}{2} u^{3}, \tag{5.4.24}
\end{equation*}
$$

subject to the boundary conditions,

$$
\begin{equation*}
U(0)=1, \quad U(1)=2 . \tag{5.4.25}
\end{equation*}
$$

The exact solution for this problem is given by,

$$
\begin{equation*}
U(x)=\frac{2}{2-x} \tag{5.4.26}
\end{equation*}
$$

By following the finite difference procedure of section (5.1), equation (5.4.24) can be approximated to obtain the difference equation (assuming $u=u_{1}$ ) ,

$$
\begin{equation*}
u_{i-1}-2 u_{i}+u_{i+1}=\frac{h^{2}}{2} u_{i}^{3}, i=1,2, \ldots, N, \tag{5.4.27}
\end{equation*}
$$

which can be simplified to the form

$$
\begin{equation*}
-u_{i-1}+2 u_{l}\left(t+\frac{h^{2}}{4} u_{i}^{2}\right)-u_{i+1}=0, i=1,2, \ldots, N \tag{5.4.28}
\end{equation*}
$$

The boundary conditions are replaced by the values,

$$
\begin{equation*}
u(0)=1, \quad u(N+1)=2, \tag{5.4.29}
\end{equation*}
$$

where,

$$
h=\frac{1}{N+1}
$$

Equation (5.4.28) can be written as:

$$
\begin{equation*}
-g_{i} u_{i-1}+2 u_{i}-g_{i} u_{i+1}=0, i=1,2, \ldots, N \tag{5.4.30}
\end{equation*}
$$

and

$$
\begin{equation*}
g_{i}=\frac{4}{4+h^{2} u_{i}^{2}} \quad, i=1,2, \ldots, N \tag{5.4.31}
\end{equation*}
$$

The system (5.4.30) can be written in matrix notation as,

$$
\begin{equation*}
\mathrm{A} \underline{u}=\underline{\mathrm{b}} \tag{5.4.32a}
\end{equation*}
$$

or,

We now split the matrix $A$ into two matrices $\bar{G}_{1}$ and $\bar{G}_{2}$ which have the form,

(5.4.33a)

if $N$ is even, and,


if N is odd. Now, for the case N even, we have,

$$
\begin{array}{ll}
G_{1}^{(i)}=\left[\begin{array}{ll}
1 & -g_{2 i-1} \\
-g_{2 i} & 1
\end{array}\right], i=1,2, \ldots, \frac{N}{2}, \\
G_{2}^{(i)}=\left[\begin{array}{ll}
1 & -g_{2 i} \\
-g_{2 i+1} & 1
\end{array}\right] \quad, i=1,2, \ldots, \frac{N}{2}-1 .
\end{array}
$$

By applying the A.G.E. method, $u^{\left(k+\frac{1}{2}\right)}$ and $u^{(k+1)}$ can be determined successively by,

$$
\begin{align*}
& \underline{u}^{\left(k+\frac{1}{2}\right)}=\left(\bar{G}_{1}+r I\right)^{-1}\left[\underline{b}-\left(\bar{G}_{2}-r I\right) \underline{u}^{(k)}\right], \\
& \underline{u}^{(k+1)}=\left(\bar{G}_{2}+r I\right)^{-1}\left[\underline{b}-\left(\bar{G}_{1}-r I\right) \underline{u}^{\left(k+\frac{1}{2}\right)}\right], \tag{5.4.35}
\end{align*}
$$

where $r$ is the iteration parameter.
Clearly, $\left(\overline{\mathrm{G}}_{1}+r \mathrm{I}\right),\left(\overline{\mathrm{G}}_{2}+r \mathrm{I}\right),\left(\overline{\mathrm{G}}_{1}-r \mathrm{I}\right)$ and $\left(\overline{\mathrm{G}}_{2}-r I\right)$ can be determined and $\left(\vec{G}_{1}+r I\right)^{-1},\left(\bar{G}_{2}-r I\right)^{-1}$ are easily invertible, as shown below,

where $\alpha=1+x$,

where $\quad d_{i}=\frac{1}{\alpha^{2}-g_{2 i-1} g_{2 i}} \quad, \quad i=1,2, \ldots \frac{N}{2}$.
where $d_{i}^{\prime}=\frac{1}{\alpha^{2}-g_{2 i} g_{2 i+1}}, i=1,2, \ldots, \frac{N}{2}-1$.

(5.4.40)
where $\beta=1-r$.

It is clear from (5.4.32b) that, $\underline{b}=\left(g_{1} u_{0}, 0, \ldots, 0,0, g_{N}{ }^{u}{ }_{N+1}\right)^{T}$ then


Now, equation (5.4.35) can be written as,


Problem II was solved on the NEPTUNE system using the strategy of the parallel A.G.E. method with both synchronous and asynchronous approaches. In our implementation, the size of the problem was taken to be $h^{-1}=25,37,39,61$ and 73 and the accuracy tolerance value $(\varepsilon)$ equal to $\left(1 \times 10^{-6}\right)$. Again, in all these parallel implementations the number of parallel paths is always equal to the number of available processors and the optimal iteration parameter $(r)$ (equation (5.3.28))
was obtained from the experiments by choosing the one that gives the best execution time.

Table 5.15 represents the results obtained from the implementation of Problem II using the parallel A.G.E. method (strategy I) using the asynchronous approach for different mesh sizes. From that table it can be noticed that the time needed for the problem to converge is increased as the number of mesh points is increased. This is due to the greater number of computational operations required to solve the problem.

Also it can be seen that the best efficiency results (speed-ups, etc.) is obtained when the problem size is equal to 60 . While for the same sizes as used in the asynchronous approach, the results of the implementation of Problem II using synchronous parallel A.G.E. method was shown in Table 5.16. From that table it is clear that the time required for the problem to converge is increased as the problem size is increased and this again is due to the more computational operations required. Also, the best efficiency (speed-up) result is obtained when the size of the problem is equal to 60 .

It can be noticed from Tables 5.15 and 5.16 that the time required for the problem to converge in the case of evaluating the points using the asynchronous approach (Table 5.15) is less than that of the synchronous approach (Table 5.16) and this is due to the synchronisation overheads needed after each iteration in the synchronous implementation. Also from these two tables it is clear that better efficiency can be obtained by using the asynchronous approach rather than the synchronous approach. This is because the speed-up ratios of the asynchronous implementation is higher than that of a synchronous one. So we summarise briefly that to solve problem II using the first strategy of the parallel
A.G.E. method the asynchronous implementation gives better results in both the time needed for the problem to converge and the speed-up ratios of the processors. This is due to the synchronisation overheads incurred by the system in the synchronisation programming implementation. The results obtained from the implementation of problem II (non-linear problem) agree with that obtained from problem I (the linear problem), i.e. in both the synchronous and asynchronous implementations in that better results are obtained with the asynchronous approach rather than a synchronous approach.

Because of the way in which algorithm II was implemented, the computational time carried out by each processor to evaluate any component is approximately the same. Besides the computational time in both the synchronous and asynchronous implementation there are an extra overhead incurred by the system which degrades the parallel algorithm performance. These overheads are the generation of the parallel paths and the synchronisation at the end of each iteration. These overheads may become important, as an example, from the results in Tables 5.15 and 5.16 and the case when the size of the problem is equal to 60 using 4 processors. In the asynchronous implementation the problem converges after 605 iterations using a total of 4 parallel paths by the first processor and 2 parallel paths by each other processor. While in the case of the synchronous implementation, the same problem requires 585 iterations to converge using a total of 1160 parallel paths by the first processor and 586 parallel paths by each other processor, since a synchronisation process was used after each iteration. Thus, it is clear the effectiveness of the overheads on the performance of the algorithm and specially in the case of the synchoonous implementation. Hence as in problem 1, we can say that for the above reason the use of the
asynchronous approach is better suited for the MIMD computer which agrees with what was obtained from the experimental results.

As in problem $I$, for comparison reasons the parallel versions of the Jacobi, Gauss-Seidel and Non-Linear Over-Relaxation (N.L.O.R.) iterative methods (See Chapter 4) are implemented on the NEPTUNE system and used to solve problem II. In these methods the number of parallel paths was taken to be equal to the number of available processors and the accuracy tolerance value ( $\varepsilon$ ) is again taken to be equal to $\left(1 \times 10^{-6}\right)$. In the parallel N.L.O.R. method the optimal $\varepsilon$ is obtained from the experimental results by choosing the value which gives the shortest time. The synchronous and asynchronous results for problem II from the implementation of the parallel Jacobi method on the NEPTUNE system are shown in Tables 5.17 and 5.18 respectively and the results from the parallel Gauss-Seidel implementation for the same problem with synchronous and asynchronous approaches are shown in Tables 5.19 and 5.20. While the results obtained from the parallel synchronous and asynchronous N.L.O.R. method are shown in Tables 5.21 and 5.22 respectively.

By comparing these results with those obtained from the parallel
A.G.E. method using strategy I with the synchronous and asynchronous approaches (Tables 5.15 and 5.16 ), it is clear that the time needed for problem II to converge using the parallel A.G.E. method gives better results in the case of the parallel Jacobi and Gauss-Seidel methods. The number of iterations in the parallel Jacobi and Gauss-Seidel methods are higher than that of the parallel A.G.E. method, which means more total computational operations are required to achieve a solution.

| Size | $\varepsilon$ | $r$ | P | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $1 \times 10^{-6}$ | 0.31 | 1 | 30.89 | 82 | 1.0 | 4 | 1 |
|  |  | 0.30 | 2 | 16.06 | 85 | 1.92341 | 4,2 | 1,1 |
|  |  | 0.31 | 3 | 11.14 | 89 | 2.77289 | 4,2,2 | 1,1,1 |
|  |  | 0.30 | 4 | 8.65 | 90 | 3.57109 | 4,2,2,2 | 1,1,1,1 |
| 36 | $1 \times 10^{-6}$ | 0.33 | 1 | 106.90 | 188 | 1.0 | 4 | 1 |
|  |  | 0.33 | 2 | 55.10 | 194 | 1.94012 | 4,2 | 1,1 |
|  |  | 0.30 | 3 | 37.57 | 197 | 2.84536 | 4,2,2 | 1,1,1 |
|  |  | 0.35 | 4 | 27.75 | 213 | 3.85225 | 4,2,2,2 | 1,1,1,1 |
| 48 | $1 \times 10^{-6}$ | 0.30 | 1 | 237.15 | 313 | 1.0 | 4 | 1 |
|  |  | 0.35 | 2 | 121.79 | 320 | 1.94720 | 4,2 | 1,1 |
|  |  | 0.31 | 3 | 83.11 | 325 | 2.85345 | 4,2,2 | 1,1,1 |
|  |  | 0.32 | 4 | 61.40 | 327 | 3.86238 | 4,2,2,2 | 1,1,1,1 |
| 60 | $1 \times 10^{-6}$ | 0.39 | 1 | 556.19 | 585 | 1.0 | 4 | 1 |
|  |  | 0.39 | 2 | 282.33 | 592 | 1.97000 | 4,2 | 1,1 |
|  |  | 0.40 | 3 | 191.03 | 603 | 2.91153 | 4,2,2 | 1,1,1 |
|  |  | 0.37 | 4 | 143.33 | 605 | 3.88049 | 4,2,2,2 | 1,1,1,1 |
| 72 | $1 \times 10^{-6}$ | 0.37 | 1 | 890.91 | 781 | 1.0 | 4 | 1 |
|  |  | 0.40 | 2 | 455.36 | 800 | 1.95650 | 4,2 | 1,1 |
|  |  | 0.35 | 3 | 307.28 | 810 | 2.89934 | 4,2,2 | 1,1,1 |
|  |  | 0.35 | 4 | 231.49 | 817 | 3.85342 | 4,2,2,2 | 1,1,1,1 |

TABLE 5.15: The results from the parallel A.G.E. method using strategy I with an asynchronous approach (Problem II)

400

| Size | $\varepsilon$ | $r$ | P | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $1 \times 10^{-6}$ | $\begin{gathered} 0.31 \\ 0.31 \\ 0.37 \\ 0.31 \\ \hline \end{gathered}$ | 2 3 4 | $\begin{array}{r} 31.12 \\ 16.70 \\ 11.81 \\ 9.30 \\ \hline \end{array}$ | $\begin{aligned} & 82 \\ & 82 \\ & 82 \\ & 82 \\ & \hline \end{aligned}$ | $\begin{array}{\|c\|} 1.0 \\ 1.86347 \\ 2.63506 \\ 3.34624 \\ \hline \end{array}$ | $\begin{array}{\|c\|} 166 \\ 166,83 \\ 166,83,83 \\ 166,83,83,83 \\ \hline \end{array}$ | $\begin{gathered} 82 \\ 82,82 \\ 82,82,82 \\ 82,82,82,82 \\ \hline \end{gathered}$ |
| 36 | $1 \times 10^{-6}$ | $\begin{gathered} 0.33 \\ 0.33 \\ 0.33 \\ 0.33 \end{gathered}$ | 1 2 3 4 | $\begin{array}{r} 107.31 \\ 56.63 \\ 39.18 \\ 29.99 \end{array}$ | $\begin{aligned} & 188 \\ & 188 \\ & 188 \\ & 188 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.89493 \\ 2.73890 \\ 3.57819 \end{gathered}$ | $\left.\begin{array}{\|r\|} 378 \\ 378,189 \\ 378,189,189 \\ 378,189,189 \\ 189 \end{array} \right\rvert\,$ | $\begin{array}{\|r} 188 \\ 188,188 \\ 188,188,188 \\ 188,188,188, \\ 188 \end{array}$ |
| 48 | $1 \times 10^{-6}$ | $\begin{aligned} & 0.3 \\ & 0.3 \\ & 0.3 \\ & 0.3 \end{aligned}$ | 1 2 3 4 | $\begin{array}{r} 237.94 \\ 124.57 \\ 85.42 \\ 65.65 \end{array}$ | $\begin{aligned} & 313 \\ & 313 \\ & 313 \\ & 313 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.91009 \\ 2.78553 \\ 3.62437 \end{gathered}$ | 628 628,314 $628,314,314$ $628,314,314$, 314 | $\begin{gathered} 313 \\ 313,313 \\ 313,313,313, \\ 313,313,313, \\ 313 \end{gathered}$ |
| 60 | $1 \times 10^{-6}$ | $\begin{aligned} & 0.39 \\ & 0.39 \\ & 0.35 \\ & 0.35 \end{aligned}$ | 1 2 3 4 | $\begin{aligned} & 558.91 \\ & 289.76 \\ & 197.83 \\ & 153.98 \end{aligned}$ | $\begin{aligned} & 585 \\ & 585 \\ & 585 \\ & 585 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.92887 \\ 2.82520 \\ 3.62976 \end{gathered}$ | 1160 1160,586 $1160,586,586$ 1160,585, 586,586 | 585 585,585 $585,585,585$ $585,585,585$, 585 |
| 72 | $1 \times 10^{-6}$ | $\begin{aligned} & 0.37 \\ & 0.36 \\ & 0.36 \\ & 0.36 \end{aligned}$ | 1 2 3 4 | $\begin{aligned} & 892.80 \\ & 471.22 \\ & 321.43 \\ & 247.55 \end{aligned}$ | $\begin{aligned} & 781 \\ & 781 \\ & 781 \\ & 781 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.89466 \\ 2.77759 \\ 3.60654 \end{gathered}$ | 1564 1564,782 $1564,782,782$ $1564,782,782$ 782 | 781 781,781 $781,781,781$ $781,781,781$, 781 |

TABLE 5.16: The results from the parallel A.G.E. method using strategy I with the synchronous approach (Problem II)

| Size | $\varepsilon$ | P | $\begin{aligned} & \text { Time } \\ & \text { (seconds) } \end{aligned}$ | No. of iterations | Speed-up | Total No. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $1 \times 10^{-6}$ | 1 2 3 4 | $\begin{aligned} & 388.05 \\ & 206.37 \\ & 142.78 \\ & 114.80 \end{aligned}$ | $\begin{aligned} & 873 \\ & 873 \\ & 873 \\ & 873 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.88036 \\ 2.71782 \\ 3.38023 \end{gathered}$ | $\begin{array}{\|c\|} 1748 \\ 1748,874 \\ 1748,874,874 \\ 1748,874,874 \\ 874 \end{array}$ | $\begin{gathered} 873 \\ 873,873 \\ 873,873,873 \\ 873,873,873, \\ 873 \end{gathered}$ |
| 36 | $1 \times 10^{-6}$ | 1 2 3 4 | $\begin{array}{r} 1108.43 \\ 582.40 \\ 411.23 \\ 318.16 \end{array}$ | $\begin{aligned} & 1732 \\ & 1733 \\ & 1732 \\ & 1730 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.90321 \\ 2.69540 \\ 3.48388 \end{gathered}$ | $\begin{gathered} 3466 \\ 3468,1734 \\ 3466,1733,1733 \\ 3462,1731, \\ 1731,1731 \\ \hline \end{gathered}$ | $\begin{gathered} 1732 \\ 1733,1733 \\ 1732,1732,1732 \\ 1730,1730, \\ 1730,1730 \\ \hline \end{gathered}$ |
| 48 | $1 \times 10^{-6}$ | 1 2 3 4 | $\begin{array}{r} 2358.57 \\ 1236.57 \\ 863.08 \\ 623.23 \end{array}$ | $\begin{aligned} & 2849 \\ & 2849 \\ & 2849 \\ & 2849 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.90735 \\ 2.73274 \\ 2.78443 \end{gathered}$ | ```5700 5700,2850 5700,2850, 2850 5700,2850, 2850,2850``` | $\begin{array}{r} 2849 \\ 2849,2849 \\ 2849,2849 \\ 2849 \\ 2849,2849 \\ 2849,2849 \end{array}$ |
| 60 | $1 \times 10^{-6}$ | 1 2 3 | $\begin{aligned} & 4257.64 \\ & 2216.39 \\ & 1446.17 \\ & 1153.15 \end{aligned}$ | 4169 <br> 4167 <br> 3840 <br> 3887 | $\begin{gathered} 1.0 \\ 1.92098 \\ 2.94408 \\ 3.69218 \end{gathered}$ | $\begin{gathered} 8340 \\ 8336,4168 \\ 7682,3841, \\ 3841 \\ 7776,3888, \\ 3888,3888 \end{gathered}$ | $\begin{array}{r} 4169 \\ 4167,4167 \\ 3840,3840 \\ 3840 \\ 3887,3887 \\ 3887,3887 \end{array}$ |
| 72 | $1 \times 10^{-6}$ | 1 2 3 | $\begin{aligned} & 6878.68 \\ & 3636.74 \\ & 2539.53 \\ & 1964.45 \end{aligned}$ | $\begin{aligned} & 5666 \\ & 5666 \\ & 5666 \\ & 5666 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.89144 \\ 2.70864 \\ 3.50158 \end{gathered}$ | $\begin{array}{r} 11334 \\ 11334,5667 \\ 11334,5667 \\ 5667 \\ 11334,5667 \\ 5667,5667 \end{array}$ | ```5 6 6 6 5666,5666 5666,5666, 5666 5666,5666, 5666,5666``` |

TABLE 5.17: The results from the parallel Jacobi iterative method using the synchronous approach (Problem II)

| Size | $\varepsilon$ | P | $\begin{aligned} & \text { Time } \\ & \text { (seconds) } \end{aligned}$ | No. of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $1 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{aligned} & 368.32 \\ & 205.65 \\ & 137.29 \\ & 103.68 \\ & \hline \end{aligned}$ | $\begin{aligned} & 873 \\ & 866 \\ & 880 \\ & 899 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.71900 \\ 2.68279 \\ 3.55247 \\ \hline \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 36 | $1 \times 10^{-6}$ | $\begin{aligned} & 1 \\ & 2 \\ & 3 \\ & 4 \end{aligned}$ | $\begin{array}{r} 1052.91 \\ 590.78 \\ 415.36 \\ 303.74 \end{array}$ | $\begin{aligned} & 1732 \\ & 1650 \\ & 1813 \\ & 1765 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.78224 \\ 2.53493 \\ 3.46648 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 48 | $1 \times 10^{-6}$ | 1 2 3 4 | $\begin{array}{r} 2240.14 \\ 1267.02 \\ 881.99 \\ 654.94 \\ \hline \end{array}$ | $\begin{aligned} & 2849 \\ & 2660 \\ & 2889 \\ & 2920 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.76804 \\ 2.53987 \\ 3.42037 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 60 | $1 \times 10^{-6}$ | 1 2 3 4 | $\begin{aligned} & 4040.33 \\ & 2284.03 \\ & 1592.40 \\ & 1230.85 \end{aligned}$ | $\begin{aligned} & 4169 \\ & 3869 \\ & 4218 \\ & 4551 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.76895 \\ 2.53726 \\ 3.28255 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 72 | $1 \times 10^{-6}$ | $1$ | $\begin{aligned} & 6521.2 \\ & 3856.73 \\ & 2541.67 \\ & 1937.01 \end{aligned}$ | $\begin{aligned} & 5666 \\ & 5599 \\ & 5651 \\ & 5921 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.69086 \\ 2.56572 \\ 3.36663 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |

TABLE 5.18: The results from the parallel Jacobi iterative method using the asynchronous approach (Problem II)

| Size | $\varepsilon$ | P | $\begin{aligned} & \text { Time } \\ & \text { (seconds) } \end{aligned}$ | No. of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $1 \times 10^{-6}$ | 1 2 3 4 | $\begin{gathered} 215.97 \\ 115.120 \\ 81.97 \\ 64.61 \end{gathered}$ | $\begin{aligned} & 442 \\ & 442 \\ & 442 \\ & 441 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.87604 \\ 2.63474 \\ 3.34267 \end{gathered}$ | $\begin{gathered} 886 \\ 886,443 \\ 886,443,443 \\ 884,442,442, \\ 442 \end{gathered}$ | $\begin{gathered} 442 \\ 442,442 \\ 442,442,442 \\ 441,441,441 \\ 441 \end{gathered}$ |
| 36 | $1 \times 10^{-6}$ | 1 2 3 4 | $\begin{aligned} & 643.48 \\ & 336.72 \\ & 237.22 \\ & 183.67 \end{aligned}$ | $\begin{aligned} & 880 \\ & 879 \\ & 878 \\ & 878 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.91102 \\ 2.71259 \\ 3.50346 \end{gathered}$ | $\begin{gathered} 1762 \\ 1760,880 \\ 1758,879,879 \\ 1758,879,879 \\ 879 \\ \hline \end{gathered}$ | $\begin{gathered} 880 \\ 879,879 \\ 878,878,878 \\ 878,878,878 \\ 878 \\ \hline \end{gathered}$ |
| 48 | $1 \times 10^{-6}$ | 1 2 3 4 | $\begin{array}{r} 1393.41 \\ 723.98 \\ 508.67 \\ 391.79 \end{array}$ | $\begin{aligned} & 1432 \\ & 1431 \\ & 1432 \\ & 1432 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.92465 \\ 2.73932 \\ \\ 3.55652 \end{gathered}$ | $\begin{gathered} 2866 \\ 2864,1432 \\ 2866,1433, \\ 1433 \\ 2866,1433, \\ 1433,1433 \end{gathered}$ | $\begin{gathered} 1432 \\ 1431,1431 \\ 1432,1432, \\ 1432 \\ 1432,1432,1432, \\ 1432 \end{gathered}$ |
| 60 | $1 \times 10^{-6}$ | 1 2 3 | $\begin{array}{r} 2540.06 \\ 1317.62 \\ 923.77 \\ 707.62 \end{array}$ | $\begin{aligned} & 2094 \\ & 2094 \\ & 2094 \\ & 2094 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.92776 \\ 2.74967 \\ 3.58958 \end{gathered}$ | $\begin{array}{r} 4190 \\ 4190,2095 \\ 4190,2095 \\ 2095 \\ 4190,2095, \\ 2095,2095 \end{array}$ | $\begin{gathered} 2094 \\ 2094,2094 \\ 2094,2094,2094 \\ 2094,2094,2094, \\ 2094 \end{gathered}$ |
| 72 | $1 \times 10^{-6}$ | 1 2 3 | $\begin{aligned} & 4108.16 \\ & 2124.06 \\ & 1495.25 \\ & 1139.58 \end{aligned}$ | $\begin{aligned} & 2849 \\ & 2849 \\ & 2848 \\ & 2848 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.93411 \\ 2.74747 \\ 3.60498 \end{gathered}$ | $\begin{gathered} 5700 \\ 5700,2850 \\ 5698,2849 \\ 2849 \\ 5698,2849 \\ 2849,2849 \end{gathered}$ | $\begin{gathered} 2849 \\ 2849,2849 \\ 2848,2848,2848 \\ 2848,2848,2848, \\ 2848 \end{gathered}$ |

TABLE 5.19: The results from the parallel Gauss-Seidel iterative method using the synchronous approach (Problem II)

| Size | $\varepsilon$ | P | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $1 \times 10^{-6}$ | \|l | $\begin{array}{r} 215.91 \\ 110.32 \\ 76.45 \\ 58.28 \\ \hline \end{array}$ | $\begin{aligned} & 442 \\ & 456 \\ & 490 \\ & 510 \\ & \hline \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.95713 \\ 2.82420 \\ 3.70470 \\ \hline \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 36 | $1 \times 10^{-6}$ | 3 <br> 4 | $\begin{aligned} & 642.89 \\ & 330.71 \\ & 225.10 \\ & 170.00 \\ & \hline \end{aligned}$ | $\begin{aligned} & 880 \\ & 907 \\ & 968 \\ & 989 \\ & \hline \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.94397 \\ 2.85602 \\ 3.78171 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 48 | $1 \times 10^{-6}$ | 1 <br> 2 <br> 3 <br> 4 | $\begin{array}{r} 1389.98 \\ 704.18 \\ 489.18 \\ 368.97 \\ \hline \end{array}$ | $\begin{aligned} & 1432 \\ & 1457 \\ & 1587 \\ & 1653 \\ & \hline \end{aligned}$ | $\begin{array}{\|c} 1.0 \\ 1.97390 \\ 2.84145 \\ 3.76719 \\ \hline \end{array}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 60 | $1 \times 10^{-6}$ | 1 2 3 4 4 | $\begin{array}{r} 2533.97 \\ 1277.62 \\ 888.22 \\ 667.07 \\ \hline \end{array}$ | $\begin{aligned} & 2094 \\ & 2126 \\ & 2322 \\ & 2413 \\ & \hline \end{aligned}$ | $\begin{array}{\|c\|} 1.0 \\ 1.98335 \\ 2.85286 \\ 3.79866 \\ \hline \end{array}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \\ \hline \end{gathered}$ |
| 72 | $1 \times 10^{-6}$ | 1 2 3 4 | 4104.43 <br> 2074.51 <br> 1445.29 <br> 1070.06 | $\begin{aligned} & 2849 \\ & 2892 \\ & 3188 \\ & 3192 \end{aligned}$ | $\begin{gathered} 1.0 \\ 1.97851 \\ 2.83987 \\ 3.83570 \end{gathered}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |

TABLE 5.20: The results from the parallel Gauss-Seidel iterative method using the asynchronous approach (Problem II)

| Size | $\varepsilon$ | P | $\omega$ | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $1 \times 10^{-6}$ | 1 | $\begin{aligned} & 1.76 \\ & 1.76 \\ & 1.76 \\ & 1.76 \end{aligned}$ | $\begin{array}{r} 26.01 \\ 14.92 \\ 9.97 \\ 7.55 \end{array}$ | $\begin{aligned} & 51 \\ & 50 \\ & 52 \\ & 49 \end{aligned}$ | $\begin{array}{\|c} 1.0 \\ 1.74330 \\ 2.60883 \\ 3.44503 \end{array}$ | $\begin{array}{\|c} 104 \\ 102,51 \\ 106,53,53 \\ 100,50,50, \\ 50 \\ \hline \end{array}$ | $\begin{gathered} 51 \\ 50,50 \\ 52,52,52 \\ 49,49,49,49 \end{gathered}$ |
| 36 | $1 \times 10^{-6}$ | 1 | $\begin{aligned} & 1.83 \\ & 1.83 \\ & 1.83 \\ & 1.83 \end{aligned}$ | $\begin{aligned} & 58.33 \\ & 31.12 \\ & 21.16 \\ & 16.10 \end{aligned}$ | $\begin{aligned} & 77 \\ & 77 \\ & 74 \\ & 73 \end{aligned}$ | $\begin{array}{\|c\|} 1.0 \\ 1.87436 \\ 2.75662 \\ 3.62298 \end{array}$ | $\begin{array}{\|c} 156 \\ 156,78 \\ 150,75,75 \\ 148,74,74 \\ 74 \\ \hline \end{array}$ | $\begin{gathered} 77 \\ 77,77 \\ 74,74,74 \\ 73,73,73,73 \end{gathered}$ |
| 48 | $1 \times 10^{-6}$ | 1 2 3 | $\begin{aligned} & 1.87 \\ & 1.87 \\ & 1.87 \\ & 1.87 \end{aligned}$ | $\begin{aligned} & 97.78 \\ & 51.85 \\ & 36.01 \\ & 27.39 \end{aligned}$ | $\begin{array}{r} 99 \\ 98 \\ 101 \\ 97 \end{array}$ | $\begin{array}{\|c} 1.0 \\ 1.88582 \\ 2.71536 \\ \\ 3.56992 \end{array}$ | $\begin{gathered} 200 \\ 198,99 \\ 204,102, \\ 102 \\ 196,98,98, \\ 98 \\ \hline \end{gathered}$ | $\begin{gathered} 99 \\ 98,98 \\ 101,101,101 \\ 97,97,97,97 \end{gathered}$ |
| 60 | $1 \times 10^{-6}$ | 1 2 3 4 | $\begin{aligned} & 1.9 \\ & 1.9 \\ & 1.9 \\ & 1.9 \end{aligned}$ | $\begin{array}{r} 148.81 \\ 80.56 \\ 55.74 \\ 42.69 \end{array}$ | $\begin{aligned} & 123 \\ & 123 \\ & 123 \\ & 122 \end{aligned}$ | 1.0 <br> 1.8472 <br> 2.66972 <br> 3.48583 | 248 248,124 $248,124,124$ $246,123,123$, 123 | $\begin{gathered} 123 \\ 123,123 \\ 123,123,123 \\ 122,122,122, \\ 122 \end{gathered}$ |
| 72 | $1 \times 10^{-6}$ | 1 2 3 4 | $\begin{aligned} & 1.91 \\ & 1.91 \\ & 1.91 \\ & 1.91 \end{aligned}$ | $\begin{array}{r} 217.86 \\ 112.05 \\ 76.28 \\ 58.35 \end{array}$ | $\begin{aligned} & 148 \\ & 146 \\ & 145 \\ & 144 \end{aligned}$ | $\begin{array}{\|c\|} \hline 1.0 \\ 1.94431 \\ 2.85606 \\ 3.73368 \end{array}$ | $\begin{array}{\|c\|} 298 \\ 294,147 \\ 292,146,146 \\ 290,145,145 \\ 145 \end{array}$ | $\begin{gathered} 148 \\ 146,146 \\ 145,145,145 \\ 144,144,144, \\ 144 \end{gathered}$ |

TABLE 5.21: The results from the parallel N.L.O.R. iterative method using the synchronous approach (Problem II)

| Size | $\varepsilon$ | P | $\omega$ | $\begin{gathered} \text { Time } \\ \text { (seconds) } \end{gathered}$ | No. of iterations | Speed-up | Total no. of parallel paths | Effective no. of parallel paths |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 24 | $1 \times 10^{-6}$ | 1 2 3 4 | $\begin{aligned} & 1.76 \\ & 1.76 \\ & 1.76 \\ & 1.77 \\ & \hline \end{aligned}$ | $\begin{array}{r} 24.71 \\ 13.88 \\ 8.90 \\ 7.41 \\ \hline \end{array}$ | $\begin{aligned} & 51 \\ & 53 \\ & 55 \\ & 62 \\ & \hline \end{aligned}$ | $\begin{array}{\|c\|} 1.0 \\ 1.78026 \\ 2.77640 \\ 3.33468 \\ \hline \end{array}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 36 | $1 \times 10^{-6}$ | 1 2 3 4 | $\begin{aligned} & 1.83 \\ & 1.83 \\ & 1.83 \\ & 1.83 \end{aligned}$ | $\begin{aligned} & 54.33 \\ & 30.37 \\ & 18.81 \\ & 17.82 \end{aligned}$ | $\begin{aligned} & 75 \\ & 82 \\ & 77 \\ & 99 \end{aligned}$ | $\begin{array}{\|c\|} \hline 1.0 \\ 1.78894 \\ 2.88836 \\ 3.04882 \\ \hline \end{array}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 48 | $1 \times 10^{-6}$ | 1 2 3 4 | $\begin{aligned} & 1.87 \\ & 1.87 \\ & 1.87 \\ & 1.88 \\ & \hline \end{aligned}$ | $\begin{aligned} & 94.00 \\ & 53.17 \\ & 31.62 \\ & 27.16 \\ & \hline \end{aligned}$ | $\begin{array}{r} 99 \\ 111 \\ 101 \\ 119 \\ \hline \end{array}$ | $\begin{array}{\|c} 1.0 \\ 1.76791 \\ 2.97280 \\ 3.46097 \end{array}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |
| 60 | $1 \times 10^{-6}$ | 1 2 3 4 | $\begin{aligned} & 1.9 \\ & 1.9 \\ & 1.9 \\ & 1.9 \\ & \hline \end{aligned}$ | $\begin{array}{r} 142.80 \\ 72.59 \\ 50.39 \\ 41.23 \\ \hline \end{array}$ | $\begin{aligned} & 123 \\ & 126 \\ & 130 \\ & 136 \\ & \hline \end{aligned}$ | $\begin{array}{\|c\|} 1.0 \\ 1.96721 \\ 2.83390 \\ 3.46350 \\ \hline \end{array}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \\ \hline \end{gathered}$ |
| 72 | $1 \times 10^{-6}$ | 1 2 3 4 | $\begin{aligned} & 1.91 \\ & 1.91 \\ & 1.91 \\ & 1.92 \end{aligned}$ | $\begin{array}{r} 208.36 \\ 106.51 \\ 74.01 \\ 54.85 \end{array}$ | $\begin{aligned} & 147 \\ & 152 \\ & 157 \\ & 155 \end{aligned}$ | $\begin{array}{\|c\|} 1.0 \\ 1.95625 \\ 2.81530 \\ 3.79872 \end{array}$ | $\begin{gathered} 4 \\ 4,2 \\ 4,2,2 \\ 4,2,2,2 \end{gathered}$ | $\begin{gathered} 1 \\ 1,1 \\ 1,1,1 \\ 1,1,1,1 \end{gathered}$ |

TABLE 5.22: The results from the parallel N.L.O.R. iterative method using the asynchronous approach
(Problem II)


## FIGURE 5.9

The timing results of parallel Jacobi, Gauss_Seidel, N.L.O.R. and A.G.E. method using the synchronous approch for problem size equel to 60


The timing results of parallel Jacobi, Gauss_Seidel, N.L.O.R. and A.G.E. methods using the asynchronous approch for problem size equel to 60

While the parallel N.L.O.R. method gives the shortest timing results than those of the parallel A.G.E. method and this is because the number of iterations needed for problem II to converge in the case of parallel A.G.E. method is higher than that of parallel N.I.O.R. method and more computational operations are required.

As an example, Figures 5.9 and 5.10 show the run time results using the parallel A.G.E., Jacobi, Gauss-Seidel and N.L.O.R. methods when the size of the problem is equal to 60 using the synchronous and asynchronous approaches respectively.

A further analysis for the parallel A.G.E. method used to solve problem II can be performed by following the steps used in problem I using the system commands XPFCLS, XPFCLN and XPFCL. The resource timings of the NEPTUNE system (Figure 4.15) and the experimental results in Tables (5.15) and (5.16) are also used.

In fact Tables (5.23) give the mean rate of access to the shared data and parallel path scheduling. It gives estimates of the potential speed-up for using $P$ processors, where $N$ represents the problem size to be solved. On the other hand, Table (5.24) illustrates the results obtained when the algorithms were run on the NEPTUNE system. The parallel control access overheads (PCO) are taken for the case $P=1$ and N is taken to be that which gives the best speed-up ratios (i.e. $N=60$ ) .

From the results in Tables (5.23) and (5.24), we notice that the parallel A.G.E. method using the asynchronous approach gives better results than that of the synchronous approach and we see that a linear speed-up has been achieved and as many as $N$ (the number of mesh points)
processors can be employed as an upper limit. From Table (5.23) the parallel A.G.E. method with the asynchronous approach has made 90 accesses to the shared data per 143 floating point operators (flops). Also from the information in Table (5.23), the static shared data access overheads in this algorithm is equal to:

$$
\frac{1}{\left(\frac{143}{90}\right)} * \frac{0.75}{720} * 100=0.06 \%
$$

This method from its program also made 1 access per ( $16{ }^{*} N_{p}$ ) flops, where $N_{P}$ is equal to the number of points in each process. This results in a parallel path access loss equal to $0.087 \%$ since the parallel path mechanism requires $\sim 1200 \mu s$.

To conclude we can say that from Problem II and from figures in Tables (5.23) and (5.24) the best results are obtained when strategy I of the parallel A.G.E. method is used with the asynchronous approach. Also the experimental results obtained from the NEPTUNE system shows that in the case of the asynchronous approach the parallel control and shared data overheads are less than that of the synchronous approach.

| Program | Processor (P) |  | Shared Data |  | Parallel Path |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: |
|  | No. | Speed-up | Access <br> rate | Overhead <br> amount | Access <br> rate | Overhead <br> amount |
| Parallel A.G.E. <br> synchronous | PSN | $0(P)$ | $91: 152$ <br> flops | $0.06 \%$ | $1:(16 *$ <br> $\left.N_{P}\right)$ <br> flops | $0.087 \%$ |
| Parallel A.G.E. <br> asynchronous | PSN | $O(P)$ | $90: 143$ <br> flops | $0.06 \%$ | $1:(16 *$ <br> $\left.N_{P}\right)$ | $0.087 \%$ |
| flops |  |  |  |  |  |  |

TABLE 5.23: Resource demands of the parallel A.G.E. algorithms

| Program | Speed-up <br> Parallel A.G.E. |  |  | 1.92887 | 2.82520 |
| :---: | :---: | :---: | :---: | :---: | :---: |
| Parchronous | 3.62976 | Shared data <br> overhead <br> syn) | Parallel control <br> overhead <br> (PCO) |  |  |
| Parallel A.G.E. <br> asynchronous | 1.97000 | 2.91153 | 3.88049 | $0.17 \%$ | $1.48 \%$ |

TABLE 5.24: Performance measurements of the parallel A.G.E. methods on the NEPTUNE system

### 5.5 CONCLUSIONS

In this chapter the parallel A.G.E. methods have been developed from the basic A.G.E. theory and the parallel versions have been implemented on the NEPTUNE system. The implemented parallel A.G.E. methods have been used to solve two types of boundary value problems. The first is of linear type while the other type is the non-linear problem. For the linear problem two strategies of the parallel A.G.E. methods have been used to solve the problem and with each strategy synchronous and asynchronous approaches have been used. For Problem I, the best results were obtained when it is solved using Strategy I with the asynchronous approach. This is due to the total computational operations in Strategy II being higher than that of Strategy $I$ and also there is the case that the old values are used while evaluating the points using Strategy II. While in the second example (the non-linear problem) only strategy $I$ was used and in all the parallel implementations the asynchronous approach gives better results than the synchronous approach and this is due to the synchronisation overheads occurring at the end of each iteration. Also from the experimental results it can be seen that the shared data overhead and the parallel control access overhead in the case of the asynchronous implementations are less than that of the synchronous implementation. Also by comparing the results obtained from both Problems I and II it can be noticed that a greater speed-up is obtained in Problem II and we can conclude that the amount of computations carried out over the total overheads in Problem II (the non-linear problem) is greater than that of Problem I (the linear problem). Also from the way in which the evaluation of the subsets is
carried out by each processor we can generally say that where the timing results are concerned it does not matter whether the algorithm is synchronously or asynchronously implemented. Since we decompose the problem into almost equal subsets and assign each one to different processors, this means that the amount of work carried out by each processor to evaluate any component is approximately the same. Finally to conclude this chapter it can be seen that the parallel A.G.E. method is suited for the parallel implementation on a MIMD computer which results in the almost linear speed-ups obtained from their implementation.

## Chapter Six

## PARALLEL SORTING AND SEARCHING ALGORITHMS

### 6.1 INTRODUCTION

Sorting is known as the process of arranging items in a predefined order. The arrangements of items is undertaken so that calculations which require data in a particular sequence can operate efficiently, so that output reports can be meaningfully presented. There are many "natural" orders, such as alphabetic ordering for a list of names, and ascending or descending values for a list of numbers. A high percentage of computer resources are utilized for sorting and it is a time consuming operation, even when a very efficient sorting algorithm is used. Many serial sorting algorithms have been developed and it was found that the optimal time required to sort N items is roughly proportional to $\mathrm{O}(\mathrm{NlogN})$ (Knuth [1973]). However, the introduction to parallel processing has added a new dimension to research on sorting algorithms. With the use of multiple processors, sorting times of $N$ items can be reduced, at least in theory to $O(\log N)$.

During the past decade, numerous results on parallel sorting have been published. In particular, Batcher's [1968] exhibited a complexity of $O\left(\log ^{2} N\right)$; later, several optimal parallel sorting algorithms, of complexity $O(\operatorname{logN})$ were developed for a theoretical parallel processor model (Hirschberg [1978]) and Preparata [1978]). The most striking property of all these algorithms is perhaps, the very large number of processors that they require. Typically, N processors are required to sort N elements.

Two factors that may affect the performance of a sorting algorithm are the number of comparisons and the conditionally exchange of data during each time unit. So parallelism may be exploited by performing more than a single comparison at a time and to move many keys
simultaneously during each unit of time. Since an optimal serial
algorithm sorts $N$ keys in time $O(N \operatorname{logN})$, the optimal speed-up would be achieved when, using $N$ processors, $N$ keys are sorted in time $O(\operatorname{logN})$.

One way of writing a parallel sort algorithm is to parallelise a well-known optimal serial algorithm. On the other hand, parallelization of straight sorting methods (one that requires $O\left(N^{2}\right)$ comparisons) seems easier, but it does not lead to very fast parallel algorithms. By performing N comparisons instead of 1 in a single time unit, the execution time can be reduced from $O\left(N^{2}\right)$ to $O(N)$. An example for this kind of parallelization is a well-known parallel version of the common bubble-sort, called the odd-even transposition sort (Knuth [1973], Baudet and Stevenson [1978]). Partial parallelization of a fast serial algorithm can also lead to a parallel algorithm of order $O(N)$. For example, the serial tree selection sort can be modified so that all the comparisons at the same level of the tree are performed in parallel (Bentley [1979]).

An improvement to the above approach has been made to achieve a higher performance. The first major improvement was reached with sorting networks, that sort $N$ numbers in time $\log ^{2}(N)$ and thus, achieve a speedup of $N / \log (N)$ (Batcher [1968]). Later, Preparata [1978] showed that the optimal bound $O(\log (N))$ and speed-up (N) can be achieved with a theoretical model of $n$ processors accessing a large shared memory. For a parallel SIMD type machine a new family of sorting algorithms has been developed by Hirschberg [1978] which shows that $N$ keys can be sorted in time $O\left(K \log N\right.$ ) with $N^{1+1 / K}$ processors, where $K$ is an arbitrary integer greater than or equal to 2 using random access capability to a common memory. Thomas and Kung [1977] presented two algorithms for sorting $\mathrm{N}^{2}$
elements on an ( $\mathrm{N} \times \mathrm{N}$ ) mesh connected processor array which requires $O(N)$ routing and comparison steps, where the best previous algorithms take a time of $O(N \operatorname{logN})$. While Nassimi and Sahni [1979] developed an $O(N)$ algorithm to sort $N^{2}$ elements on an Illiac IV-like ( $N \times N$ ) mesh connected processor array (SIMD type machine). This algorithm sorts the $\mathrm{N}^{2}$ elements into row-major order and snake-like row-major order. Another class of parallel sorting algorithm based on enumeration have been developed where $N$ elements are sorted with $O(\log N)$. Sorting is performed by computing in parallel the rank of each element, and routing the elements to the location modified by their rank. The first enumeration type parallel sorting is a modified sorting network scheme, that sorts N elements with $O\left(N^{2}\right)$ processing elements. By embedding this type of network in a more general multiprocessor model, where processors have access to a large shared memory, algorithms that are as fast, but which require only $O(N)$ processors were obtained. Muller [1975] proposed a very fast sorting network parallel algorithm which was the first to use an enumeration scheme for parallel sorting. The idea of sorting by enumeration was exploited to develop other very fast parallel sorting algorithms (Hirschberg [1978] and Preparata [1978]) which improve on Muller's result by reducing the number of processing elements.

In addition to the idea of using enumeration, optimal parallel sorting algorithms use a fast merging procedure. In a study of parallelism in the comparison problem Valiant [1975] presents an inductive algorithm that merges two sorted sequences of $n$ and $m$ elements ( $n \times m$ ) with ( $n m$ ) processors in $2 \log \operatorname{logn+O}(1)$ comparison steps. On the other hand, Gavril [1975] proposed a fast merging algorithm that solves the problem of merging two sorted sequences of length $n$ and $m$ with a
smaller number of processors $p \leqslant n \leqslant m$. This algorithm is based on binary insertion, and requires only $2 \log (n+1)+4(n / p)$ comparisons when $n=m$. Both Valiant's and Gavril's algorithms assume a shared memory model. That is, all the processors utilized can simultaneously access elements of the initial data, or intermediate computation results.

For all the parallel sorting algorithms described so far it is assumed that the problem size is limited by the number of processors available. Thus, these algorithms implicitly assume that the number of processors is very large. Typically, $N$ processors are utilized to sort N records which is impractical. However, for a general purpose sorting algorithm, it is desirable to set a limit on the number of processors available, so that the number of records that can be sorted will not be bounded by the number of processors. Furthermore, it must be possible to sort a large array with a relatively small number of processors. When $P$ processors are available, and $N$ records are to be sorted, one possibility is to distribute the N records among the P processors so that a block of $M=\lceil N / P\rceil$ records is sorted in each processor's local memory. The block residing in each processors memory constitutes a sorted sequence of length $M$ and the concatenation of these local sequences constitutes a sorted sequence of length $N$.

Algorithms to sort large arrays of files that are initially distributed across the processor's local memories, can be constructed as a sequence of block merge-split steps. During a merge-split step, a processor merges two sorted blocks of equal length (that are produced by a previous step), and splits the resulting block into a "higher" and a "lower" block, that are sent to two destination processors. Two merge-split step ways have been proposed, one is based on a 2 -way merge
(Baudet [1978]) and the other based on a bitonic merge (Hsiao [1980]). In this chapter, two sorting algorithms have been implemented in parallel on the NEPTUNE system with different approaches. These algorithms are the parallel shell sort and the parallel digit sort methods. Two parallel merging algorithms are used incorporated with the sorting algorithms, these are the 2 -way merge and the odd-even reduction merge methods. Besides the parallel sorting algorithms, two parallel searching algorithms are also implemented on the NEPTUNE system. The results of all sorting and searching 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 studied.

### 6.2 THE SORTING ALGORITHMS

In this section, two parallel sorting algorithms are implemented on the NEPTUNE system and used to sort a set of N items in ascending order. The first parallel sorting algorithm is developed from the sequential shell sort, while the second one is developed from the sequential digit sort.

Generally, the parallel sort implementation is carried out by splitting the input set $N$ into $M$ subsets which are sorted in parallel where $M$ is greater than or equal to $P$, the number of available processors. Two approaches are used to implement the parallel shell sort method. In the first approach the final sorted set is obtained by using the parallel sorting algorithm only, while in the second approach each $M$ subsets is sorted independently then followed by a parallel merge algorithm to obtain the final sorted set.

### 6.2.1 The Shell Sort Method

This method, also known as the diminishing inerement sort (Knuth [1973]) is a simple sorting algorithm that requires no extra storage was developed by D.L. Shell [1959]. It consists of a number of passes over the input (unsorted) set and in each pass it consists of a number of comparisons of two keys and an interchange is carried out if they are out of order. In this way, low keys are moved towards the beginning of the list and high keys are moved towards the end, and an ordered list will be finally produced. During the first pass, keys relatively far apart are compared, in order to move the low ones that were initially near the end of the list to the beginning, and vice versa. Subsequent passes use a steadily decreasing increment between the
compared keys. While the last increment used must be 1.
To sort the sequence $x_{1}, x_{2}, \ldots, x_{n}$ using Shell's method, a number $d_{1}$ (depending on $n$ ) is chosen and each of the subsequencies $x_{i}, x_{i}+d_{1}$, $x_{i}+2 d_{1}, \ldots, x_{i}+n_{i} d_{1}$ corresponding to $i=1,2, \ldots, d_{1}$, where $n_{i}$ is the largest integer such that $n_{i}{ }_{1} \leqslant n$, is sorted by comparing $x_{i}$ with $x_{i}+d_{1}$, and are transposed if necessary. Then $x_{i}+3 d_{1}$ is compared with $x_{i}+2 d_{1}$, and if a transposition occurs, $x_{i}+3 d_{1}$ is compared with $x_{i}+d_{1}$. Then $x_{i}+4 d_{1}$ is dropped down one position at a time into its proper place, and so on up to $x_{i}+n_{i} d_{1}$. Then a number $d_{2}<d_{1}$ is chosen and the procedure is repeated on the sequences $x_{i}, x_{i}+d_{2}, \ldots$. Then $d_{3}<d_{2}$ is chosen and the procedure finally repeated with $d_{m}=1$. This last pass is simply by comparing two adjacent elements at a time and an interchange occurs as previously if they are out of order, and so completes the sort.

The running time of Shell's method depends on the optimal sequence $d_{1}, d_{2}, \ldots, d_{m}$, which is still an open question. The one proposed by Shell [1959], is that $d_{1}=\left[\frac{n}{2}\right]$ and $d_{i}=\left[\frac{d_{i-1}}{2}\right]$, where [ $]$ means the integral part. Papernov and Stasevich [1965] suggest the form $d_{1}=2^{k}+1$, where $2^{k}<n \leqslant 2^{k+1}$, and $d_{i+1}=\left[\frac{d_{i}}{2}\right]$. There are many other suggestions for the choice of the d's and in all these methods the times required range from $O\left(n^{2}\right)$ to $O\left(n \log ^{2} n\right)$, which depends on the choice of the increment d (Knuth [1973]), Papernov and Stasevich [1965]). In general, the running time for Shell's sorting method is of order $\mathrm{N}^{3 / 2}$ as an upper limit.

Two parallel versions of the Shell sort algorithm were implemented on the NEPTUNE system. In version I, the original set of $N$ elements is partitioned into $d$ subsets, each containing the elements that are $d$ positions apart. Thus, the first d elements are allocated into the first
subset and the next $d$ elements into the second subset and so on. We assume $d \geqslant P$, where $P$ is the number of available processors and more than one subset may be assigned to each processor, where $P$ subsets can be sorted in parallel (i.e. at the same time) using the sequential algorithm described previously. In Version $I$ we choose a sequence of distance $d_{1}, d_{2}, \ldots, d_{m}$, such that $d_{m}=1$ and the sorted set is obtained when all the $d^{\prime} s$ are applied to the input set, starting with $d_{1}$ and terminated by using $d_{m}$. Also for Version $I$ of the parallel Shell sort two approaches are used for its implementation on the NEPTUNE system. In the first approach, the parallel Shell sort algorithm was programmed on the NEPTUNE system (Program 6.1) by taking $d_{1}=\left[\frac{n}{2}\right]$ and $d_{i}=\left[\frac{d_{i-1}}{2}\right]$. While in the second approach, the algorithm is programmed by taking $d_{1}=2^{k}+1$ and $d_{i}=\left[\frac{d_{i}}{2}\right]$, where $2^{k}<n \leqslant 2^{k+1}$ (Program 6.2).

For the total complexity of the Shell sort algorithm when run on both the sequential (one processor) and the parallel machine ( P processors) Papernov and Stasevich [1965] and Knuth [1975] shows that the total running time $T_{1}$ of the algorithm which is composed of both the total comparisons and the total number of exchanges is equal to,

$$
\begin{equation*}
\mathrm{T}_{1}=\mathrm{K} \mathrm{~N}^{3 / 2} \tag{6.2.1}
\end{equation*}
$$

where $K$ is a known constant and is assumed to be $O(1)$.
In our implementation of the Shell sort, the set to be sorted is partitioned into subsets and each subset is assigned to a parallel path. Thus if we generate $M$ paths (M equal to the distance of comparison) with $M \geqslant P$, then in each subset (path) $\left(\frac{N}{M}\right)$ elements are sorted. If all the paths are carried out on one processor then by applying equation (6.2.1), the total complexity for the Shell algorithm is equal to,

$$
\begin{equation*}
T_{1 S}=M \times\left(\frac{N}{M}\right)^{3 / 2} \tag{6.2.2}
\end{equation*}
$$

On the other hand, when this algorithm is run in parallel using P processors, $\left[\frac{M}{P}\right\rceil$ paths have to be carried out by each processor. Thus,

$$
\begin{align*}
T_{P S} & =\left\lceil\frac{M}{P}\right\rceil \times\left(\frac{N}{M}\right)^{3 / 2}  \tag{6.2.3}\\
& \leqslant\left(\frac{M}{P}\right) \times\left(\frac{N}{M}\right)^{3 / 2}+1 .
\end{align*}
$$

The efficiency of the Shell sort algorithm when applied on a parallel computer can be measured by calculating the speed-up ratio $S_{P S}(M)$ with M subsets. Therefore,

$$
\begin{align*}
S_{P S}(M) & =\frac{T_{1 S}}{T_{P S}}  \tag{6.2.4}\\
& =P\left(1-\frac{1}{T_{P S}}\right)
\end{align*}
$$

which is of $O(P)$. This means that the optimal linear speed-up is achieved for this implementation.

The experimental results obtained from the parallel Shell sort (Version I) with different sizes on the NEPTUNE system using the first and second approaches are shown in Tables (6.1) and (6.2) respectively. The efficiency $E_{p}$ in the tables are obtained theoretically from the formula,

$$
E_{p}=\frac{\text { Speed-up }}{P}
$$

From these results it can be noticed that in the second approach the efficiency increases as the size of the input data increases. While in the first approach the efficiency is generally the same. Within each size ( $N$ ) the efficiency decreases as the number of processors increases and this is due to the reduction in the usage of processors. It is clear from these results that the second strategy gives better results from the first one. This is because the time taken to sort the input set of numbers using the second strategy is less than that of using the first strategy and the speed-up factors of the second strategy
is higher than that of the first strategy. This is due to the way in which the distance of comparisons is chosen in the first strategy, as in the final pass of the first strategy is entered, there are two partitions, that of the even positions, and that of the odd positions. Each partition is individually sorted, but there is no meaningful order relative to each other. This condition will always occur when N is a power of 2 , because the distance is always even. Thus, there will never be any comparisons between odd-numbered elements and even-numbered elements. As a consequence, the final pass must merge two independent strings. While this is not the case in the second strategy which means fewer passes will be required and better results are obtained. It is also clear from the results in Tables (6.1) and (6.2) that the values obtained from the parallel sort algorithm is not very efficient. The factor that decisively degrades its performance is the last pass, when the increment is 1 , which dominates the running time. During this pass one of the processors assigned to the task performs the straight insertion sorting on the whole set of numbers, and the remaining processors are idle. Another factor that affects the performance of this algorithm is due to the overheads incurred by the system due to the generation of a large number of parallel paths. In this implementation the total number of generated parallel paths is equal to the total distances of comparisons ( $d_{i}$ ), which varies from pass to pass and depends on the size of the input data as shown in Tables (6.1) and (6.2). Figure (6.1) shows the graphical representation of the experimental timing results obtained from the implementation of Version $I$ using the first and second approach with data size equal to 1024.

| Size <br> (N) | No. of Processors (P) | Time (sec.) | Speed-up | Total No. of Parallel Paths | $\begin{gathered} \text { Efficiency } \\ \left(E_{p}\right) \end{gathered}$ |
| :---: | :---: | :---: | :---: | :---: | :---: |
| 256 | 1 | 3.00 | 1.0 | 265 | 1.0 |
|  | 2 | 2.07 | 1.44928 | 142,124 | 0.73 |
|  | 3 | 1.88 | 1.59575 | 102,82,83 | 0.53 |
|  | 4 | 1.72 | 1.74419 | 78,63,62,65 | 0.44 |
| 512 | 1 | 8.34 | 1.0 | 522 | 1.0 |
|  | 2 | 5.70 | 1.46316 | 272,251 | 0.73 |
|  | 3 | 5.19 | 1.60694 | 188,166,170 | 0.54 |
|  | 4 | 4.77 | 1.74843 | $\begin{gathered} 145,124,132 \\ 124 \end{gathered}$ | 0.44 |
| 768 | 1 | 13.76 | 1.0 | 777 | 1.0 |
|  | 2 | 9.52 | 1.44538 | 399,379 | 0.72 |
|  | 3 | 8.60 | 1.60 | 271,248,260 | 0.53 |
|  | 4 | 8.25 | 1.66788 | $\begin{gathered} 209,186,193 \\ 192 \end{gathered}$ | 0.42 |
| 1024 | 1 | 22.41 | 1.0 | 1035 | 1.0 |
|  | 2 | 15.88 | 1.41121 | 525,511 | 0.71 |
|  | 3 | 14.31 | 1.56604 | 361,339,337 | 0.52 |
|  | 4 | 13.60 | 1.64779 | $\begin{gathered} 273,253,253, \\ 259 \end{gathered}$ | 0.41 |

TABLE 6.1: The results of the implementation of Shell sort (Version I, first approach)

| Size <br> (N) | No. of Processors ( P ) | Time <br> (sec.) | Speed-up | Total No. of Parallel Paths | $\begin{gathered} \text { Efficiency } \\ \left(E_{p}\right) \end{gathered}$ |
| :---: | :---: | :---: | :---: | :---: | :---: |
| 256 | 1 | 2.06 | 1.0 | 142 | 1.0 |
|  | 2 | 1.32 | 1.56061 | 78,65 | 0.78 |
|  | 3 | 1.03 | 2.00 | 56,45,43 | 0.67 |
|  | 4 | 0.91 | 2.26374 | 48,33,31,33 | 0.57 |
| 512 | 1 | 4.8 | 1.0 | 272 | 1.0 |
|  | 2 | 2.96 | 1.62162 | 142,131 | 0.81 |
|  | 3 | 2.21 | 2.17195 | 100,88,86 | 0.72 |
|  | 4 | 2.0 | 2.40 | 81,63,64,67 | 0.60 |
| 768 | 1 | 8.07 | 1.0 | 530 | 1.0 |
|  | 2 | 4.87 | 1.65708 | 273,258 | 0.83 |
|  | 3 | 3.55 | 2.27324 | 184,175,173 | 0.76 |
|  | 4 | 3.22 | 2.50621 | $\begin{gathered} 145,130,127 \\ 131 \end{gathered}$ | 0.63 |
| 1024 | 1 | 11.18 | 1.0 | 530 | 1.0 |
|  | 2 | 6.72 | 1.66369 | 271,260 | 0.83 |
|  | 3 | 4.85 | 2.30516 | 189,173,170 | 0.77 |
|  | 4 | 4.32 | 2.58796 | $\begin{gathered} 147,131,127 \\ 128 \end{gathered}$ | 0.65 |

TABLE 6.2: The results of the implementation of Shell sort (Version I, second approach)


FIGURE 6.1
The experimental timing results obtained from the parallel shell sort (Version I) using the first and second strategy with data size 1024.
efficiency of the algorithm, we suggest a new version of implementing the parallel Shell sort by incorporating the merge algorithm with the sort procedure. In this version (Version II) the sort algorithm is implemented in two stages, in the first stage the sort procedure is applied, while in the second stage the merge algorithm is followed and it is always in that order. In Version II, the input set is divided into $M$ subgroups where $M \geqslant P$, and $P$ is the number of available processors, then each subgroup is assigned to a parallel path which runs simultaneously. In the first stage of this version each processor will independently sort its subgroup using the parallel shell sort algorithm. When all the subgroups are sorted, the second stage is started (the merge stage) by merging the sorted subgroups to form the final sorted list using the parallel merge algorithm.

The implementation of Version II on the NEPTUNE system is carried out by using the parallel Shell sort algorithm that gives the better results in Version I (i.e. the second approach). While two parallel merge algorithms are incorporated with the parallel Shell sort algorithm, these are the parallel 2-way merge and the parallel odd-even merge algorithms which are described below.

## The Paralle1 2-Way Merge Algorithm

In this parallel merge algorithm, the following sequential 2 -way merge procedure proposed by Knuth [1973, pp.159] is applied.

Given two ordered subsets $x_{1} \leqslant x_{2} \leqslant \cdots \leqslant x_{m}$ and $y_{1} \leqslant y_{2} \leqslant \ldots \leqslant y_{n}$ into a single set $z_{1} \leqslant z_{2} \leqslant \cdots \leqslant z_{m+n}$
(1) Set $i \nleftarrow 1, j \nleftarrow 1, k \neq 1$,
(2) If $x_{i} \leqslant y_{j}$, goto step (3), otherwise goto step (5).
(3) Set $z_{k} \leftarrow X_{i}, k \leftarrow k+1$, $i \leftarrow i+1$. If $i \leqslant m$, goto step (2).
(4) Set $\left(z_{k}, \ldots, z_{m+n}\right) \leftarrow\left(y_{i}, \ldots, y_{n}\right)$ and terminate the algorithm.
(5) Set $z_{k} \leftarrow Y_{j}, k \not k+1, j+j+1$. If $j \leqslant n$, goto step (2).
(6) Set $\left(z_{k}, \ldots, z_{m+n}\right) \leftarrow\left(x_{i}, \ldots, x_{m}\right)$ and terminate the algorithm. The parallel implementation of the 2 -way merge algorithm is carried out by, applying the above sequential algorithm on $M$ sorted subsets of size $\left(\frac{N}{M}\right)$ each. By supposing that $N$ is divisible by $M$ where $M$ is a power of 2. The parallel 2-way merge algorithm can be completed in $\log _{2} M$ steps where the parallelism is introduced within each step and not amongst the steps as shown in Figure 6.2.


FIGURE 6.2: The parallel 2-way merge algorithm

From Figure 6.2 it can be seen that 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 $\left(\frac{2 M}{N}\right)$. Also it can be realized from Figure 6.2 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 where only one processor is required.
The parallel 2-way merge is used to merge subgroups that are sorted previously using the parallel shell methods and the implementation of the parallel shell sort algorithm with the parallel 2 -way merge is programmed on the NEPTUNE system as shown in Program (6.3).

For the complexity of the parallel 2 -way merge algorithm, Evans and Yousif [1985] shows that the total complexity when one procesor is used to merge $M$ subsets with size $\left(\frac{N}{M}\right)$ each is,

$$
\begin{equation*}
T_{I M}=N \log M-M+1 \tag{6.2.5}
\end{equation*}
$$

While for the parallel implementation of the 2 -way merge using $P$ processors and $M$ subsets where $M \geqslant P$, the total complexity is,

$$
\begin{equation*}
T_{P M} \leqslant \frac{N}{P} \log \left(\frac{M}{P}\right)+\frac{2 N}{P}(P-1)-\frac{M}{P}+1+\log \left(\frac{M}{P}\right) \tag{6.2.6}
\end{equation*}
$$

Now, for Version II of the parallel shell sort algorithm using the parallel 2-way merge (Program 6.3), we can obtain the total complexity which represents both the sorting and merging complexities. The complexity of the parallel shell procedure on one processor is given by $T_{1 S}$ in equation (6.2.2), therefore, the total complexity $T_{1}$ of Program 6.3 in one processor is given by:

$$
\begin{align*}
T_{1} & =T_{1 S}+T_{1 M} \\
& =M \cdot\left(\frac{N}{M}\right)^{3 / 2}+N \log M-M+1 \tag{6.2.7}
\end{align*}
$$

To obtain the total complexity $T_{P}$ of Program 6.3 when $P$ processors are used is obtained from equations (6.2.3) and (6.2.6). Thus,

$$
\begin{align*}
T_{P} & =T_{P S}+T_{P M} \\
& \leqslant\left(\frac{M}{P}\right) \cdot\left(\frac{N}{M}\right)^{3 / 2}+\frac{N}{P} \log \left(\frac{M}{P}\right)+\frac{2 N}{P}(P-1)-\frac{M}{P}+\log \left(\frac{M}{P}\right)+2 \tag{6.2.8}
\end{align*}
$$

The speed-up ratios for the merge algorithm alone ( $S_{\text {merge }}(M)$ ) and the
total speed-up for both the sort and the merge algorithms together (Stotal $(M)$ ) can now be represented,

$$
\begin{align*}
S_{\text {merge }}(M) & =\frac{T_{1 M}}{T_{P M}} \\
& \geqslant \frac{N \log M-M+1}{\frac{N}{P} \log \left(\frac{M}{P}\right)+\frac{2 N}{P}(P-1)-\frac{M}{P}+\log \left(\frac{M}{P}\right)+1} \\
& \geqslant P\left(\frac{N \log M-M+1}{N \log \left(\frac{M}{P}\right)+2 N(P-1)-M+\log \left(\frac{M}{P}\right)+P}\right) . \tag{6.2.9}
\end{align*}
$$

At the same time, the total speed-up is measured as:

$$
\begin{align*}
S_{\text {total }}(M) & =\frac{T_{1}}{T_{p}} \\
& \geqslant \frac{M \cdot\left(\frac{N}{M}\right)^{3 / 2}+N \log M-M+1}{\left(\frac{M}{P}\right) \cdot\left(\frac{N}{M}\right)^{3 / 2}+\frac{N}{P} \log \left(\frac{M}{P}\right)+\frac{2 N}{P}(P-1)-\frac{M}{P} \log \left(\frac{M}{P}\right)+2} \\
& \geqslant P \cdot\left(\frac{M \cdot\left(\frac{N}{M}\right)^{3 / 2}+N \log M-M+1}{M \cdot\left(\frac{N}{M}\right)^{3 / 2}+N \log \left(\frac{M}{P}\right)+2 N(P-1)-M+P \log \left(\frac{M}{P}\right)+2 P}\right) . \tag{6.2.10}
\end{align*}
$$

In our implementation the input size $N$ is partitioned into $M$ subgroups with size $\left(\frac{N}{M}\right)$ each, where $M \geqslant P$ ( $P$ is the number of co-operating processors). From the results obtained in Table (6.3) it is clear that the sorting speed-up is $O(p)$. We also notice that the maximum total speed-up occurs when $M=P$ where the total speed-up is less than the "linear speed-up". This 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 synchronisation involved at the end of the parallel paths, when $M>P$. On the other hand, the merge speed-up increases as $M$ increases and the maximum speed-up occurs at $\mathrm{M}=256$ (which is the largest number of paths we generate on the NEPTUNE system in our experiments). 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 $\log \left(\frac{M}{P}\right)$ steps all the processors are active and contribute to the solution of the problem and after $\log \left(\frac{M}{P}\right)$ 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. Thus, if $M \gg P$, then $\log \left(\frac{M}{P}\right)$ is large, hence all the processors are significantly active. Although the timing results increase as $M$ increases in the merge algorithm the speed-up results are improved for large M. Table (6.4) represents the experimental results with total efficiency ( $E_{p}$ ) of Version II of parallel shell sort. From both Tables (6.3) and (6.4) it can be seen that the best total sort timing results are obtained when the input set $N$ is partitioned into $M$ subsets where $M=64$. The efficiency in Table (6.4) decreases as the number of processors increases and this is due to the reduction in the usage of the processors in each step of the 2 -way merge algorithm and sorting procedure. The inequality (6.2.10) is used to calculate the total theoretical speed-up values and these values are tabulated in Table (6.5) with its corresponding experimental results in Table (6.4). The mismatching in the results (especially for large M) of Table (6.5) is due to the fact that the inequality (6.2.10) for the theoretical speed-up does not include the parallel allocation or data communication overheads.

An alternative implementation of Version II parallel Shell sort algorithm on the NEPTUNE system is carried out by using the odd-even reduction merge instead of the 2 -way merge.

| No. of <br> Processors <br> (P) | No.of <br> Paths <br> (M) | Time for <br> Sorting <br> (sec) | Time for <br> Merging <br> (sec) | Total <br> time <br> (sec) | Speed-up <br> for <br> sorting | Speed-up <br> for <br> merging | Speed-up <br> for both <br> parts |
| :--- | :---: | :--- | :--- | :--- | :--- | :--- | :--- |
| 1 | 2 | 9.10 | 0.84 | 9.94 | 1.0 | 1.0 | 1.0 |
| 2 | 2 | 4.62 | 0.85 | 5.47 | 1.96970 | 0.98824 | 1.81718 |
| 1 | 4 | 7.45 | 1.66 | 9.11 | 1.0 | 1.0 | 1.0 |
| 2 | 4 | 3.79 | 1.29 | 5.08 | 1.96570 | 1.28682 | 1.79331 |
| 4 | 4 | 1.94 | 1.27 | 3.21 | 3.84021 | 1.30709 | 2.83801 |
| 1 | 8 | 6.16 | 2.49 | 8.65 | 1.0 | 1.0 | 1.0 |
| 2 | 8 | 3.14 | 1.70 | 4.84 | 1.96178 | 1.46471 | 1.78719 |
| 4 | 256 | 2.2 | 1.61 | 1.49 | 3.10 | 3.82609 | 1.67114 |

TABLE 6.3: The experimental results of shell sort algorithm using the 2 -way merge algorithm (Version II) for input size 1024.

| $\begin{aligned} & \text { Size } \\ & \text { (N) } \end{aligned}$ | Processors <br> (P) | Parallel Paths <br> (M) | Total Time (sec) | Total <br> Speed-up | Total <br> parallel <br> paths for <br> sorting | Total <br> parallel <br> paths for <br> merging | $\begin{aligned} & \text { Total } \\ & \text { efficiency } \\ & \left(E_{p}\right) \end{aligned}$ |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1024 | 1 | $\begin{array}{r} 2 \\ 4 \\ 8 \\ 16 \\ 32 \\ 64 \\ 128 \\ 256 \\ \hline \end{array}$ | $\begin{aligned} & 9.94 \\ & 9.11 \\ & 8.65 \\ & 8.01 \\ & 7.69 \\ & 7.49 \\ & 7.52 \\ & 7.75 \end{aligned}$ | 1.0 <br> 1.0 <br> 1.0 <br> 1.0 <br> 1.0 <br> 1.0 <br> 1.0 <br> 1.0 | $\begin{array}{r} 5 \\ 7 \\ 11 \\ 19 \\ 35 \\ 67 \\ 131 \\ 259 \end{array}$ | $\begin{array}{r} 4 \\ 7 \\ 12 \\ 21 \\ 38 \\ 71 \\ 136 \\ 265 \end{array}$ | 1.0 <br> 1.0 <br> 1.0 <br> 1.0 <br> 1.0 <br> 1.0 <br> 1.0 <br> 1.0 |
|  | 2 | $\begin{array}{r} 2 \\ 4 \\ \cdot 8 \\ 16 \\ 32 \\ 64 \\ 128 \\ 256 \end{array}$ | 7.77 <br> 5.08 <br> 4.84 <br> 4.52 <br> 4.38 <br> 4.28 <br> 4.33 <br> 4.49 | $\begin{aligned} & 1.81718 \\ & 1.79331 \\ & 1.78719 \\ & 1.77212 \\ & 1.75571 \\ & 1.75000 \\ & 1.73672 \\ & 1.72606 \end{aligned}$ | $\begin{array}{\|c} 4,2 \\ 5,3 \\ 7,5 \\ 11,9 \\ 19,17 \\ 35,33 \\ 68,64 \\ 134,126 \end{array}$ | $\begin{gathered} \hline 4,1 \\ 6,2 \\ 9,4 \\ 14,8 \\ 23,16 \\ 40,32 \\ 73,64 \\ 138,128 \end{gathered}$ | 0.91 0.90 0.89 0.89 0.88 0.88 0.87 0.86 |
|  | 3 | $\begin{array}{r} 8 \\ 16 \\ 32 \\ 64 \\ 128 \\ 256 \end{array}$ | $\begin{aligned} & 3.86 \\ & 3.68 \\ & 3.60 \\ & 3.52 \\ & 3.54 \\ & 3.77 \\ & \hline \end{aligned}$ | $\begin{aligned} & \hline 2.24093 \\ & 2.17663 \\ & 2.13611 \\ & 2.12784 \\ & 2.12429 \\ & 2.05570 \end{aligned}$ | $\begin{aligned} & 6,3,4 \\ & 8,6,7 \\ & 14,12,11 \\ & 25,22,22 \\ & 46,44,43 \\ & 92,84,85 \end{aligned}$ | $\begin{aligned} & 9,3,2 \\ & 13,5,5 \\ & 19,12,9 \\ & 32,21,20 \\ & 55,42,41 \\ & 100,83,84 \end{aligned}$ | $\begin{aligned} & 0.75 \\ & 0.73 \\ & 0.71 \\ & 0.71 \\ & 0.71 \\ & 0.69 \end{aligned}$ |
|  | 4 | 4 | 3.21 | 2.83801 | 4,2,2,2 | 6,1,1,2 | 0.71 |
|  |  | 8 | 3.10 | 2.79032 | 5,3,3,3 | 8,2,2,3 | 0.70 |
|  |  | 16 | 2.94 | 2.72449 | 7,5,5,5 | 11,4,4,5 | 0.68 |
|  |  | 32 | 2.87 | 2.67944 | 11,9,9,9 | 16,8,8,9 | 0.67 |
|  |  | 64 | 2.86 | 2.61888 | $\begin{gathered} 19,17,17, \\ 17 \end{gathered}$ | $\begin{gathered} 25,17,16 \\ 16 \end{gathered}$ | 0.65 |
|  |  | 128 | 2.88 | 2.61111 | $\begin{gathered} 35,33,33, \\ 33 \end{gathered}$ | $\begin{gathered} 42,32,32, \\ 33 \end{gathered}$ | 0.65 |
|  |  | 256 | 2.98 | 2.60067 | $\begin{gathered} 69,65,63 \\ 65 \end{gathered}$ | $\begin{gathered} 76,74,63 \\ 65 \end{gathered}$ | 0.65 |

TABLE 6.4: The experimental results with total efficiency of parallel shell sort (Version II with 2-way merge)

| $\begin{aligned} & \text { Size } \\ & (\mathrm{N}) \end{aligned}$ | Processors <br> (P) | Paths <br> (M) | Theoretical Speed-up | Experimental Speed-up |
| :---: | :---: | :---: | :---: | :---: |
| 1024 | 2 | 2 4 8 16 32 64 128 256 | 1.86172 1.81383 1.75518 1.68778 1.61428 1.54229 1.47623 1.41826 | $\begin{aligned} & 1.81718 \\ & 1.79331 \\ & 1.78719 \\ & 1.77212 \\ & 1.75571 \\ & 1.75000 \\ & 1.73672 \\ & 1.72606 \\ & \hline \end{aligned}$ |
|  | 3 | $\begin{array}{r} 8 \\ 16 \\ 32 \\ 64 \\ 128 \\ 256 \end{array}$ | $\begin{aligned} & 2.32737 \\ & 2.16763 \\ & 2.00658 \\ & 1.86000 \\ & 1.73000 \\ & 1.62203 \\ & \hline \end{aligned}$ | $\begin{aligned} & 2.24093 \\ & 2.17663 \\ & 2.13611 \\ & 2.12784 \\ & 2.14245 \\ & 2.05570 \\ & \hline \end{aligned}$ |
|  | 4 | $\begin{array}{r} 4 \\ 8 \\ 16 \\ 32 \\ 64 \\ 128 \\ 256 \end{array}$ | $\begin{aligned} & 3.01749 \\ & 2.77249 \\ & 2.51825 \\ & 2.27487 \\ & 2.06010 \\ & 1.88280 \\ & 1.73825 \end{aligned}$ | $\begin{aligned} & 2.83801 \\ & 2.79032 \\ & 2.72449 \\ & 2.67944 \\ & 2.61888 \\ & 2.61111 \\ & 2.60067 \end{aligned}$ |

TABLE 6.5: The theoretical and experimental values of the total speed-up of Program 6.3.

## The Parallel Odd-Even Merge Algorithm

This version of the odd-even merge algorithm is an extension to that of Baudet and Stevenson [1978] algorithm which it self is an improvement to the original odd-even merge algorithm that is based on Bacher's 0-1 merge (Batcher [1978]). The algorithm is described as follows:

The input set of $N$ elements is partitioned into $M$ subsets of size $\left(\frac{N}{M}\right)$ each. Each of the $M$ subsets are sorted using the parallel shell procedure, so the subsets become sorted within themselves but not amongst each other. Now these sorted 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. The graphical representation of this merge algorithm is shown in Figure 6.3 for $M=8$.


FIGURE 6.3: The parallel odd-even merge algorithm

Figure 6.3 illustrates that odd numbered subsets are merged with the even numbered subsets in the odd numbered steps of the algorithm. While in the even numbered steps, the even numbered subsets are merged with the odd 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 6.3. The final sorted set can be obtained at the end of final step (step M) by combining the sorted subsets.

The implementation of the parallel shell sort algorithm (Version II) was carried out by incorporating the parallel odd-even merge algorithm with the parallel shell sort procedure and the program of this implementation is given in Program 6.4. Table 6.6 shows the experimental results obtained from Program 6.4 when it runs on the NEPTUNE system using a data size equal to 1024.

For the complexity of the parallel odd-even merge algorithm, Yousif and Evans [1985a] shows that the complexity of the odd-even merge algorithm when run on one processor is equal to,

$$
\begin{equation*}
C_{1 M}=N(M-1)-\frac{1}{2} M(M-1) \tag{6.2.11}
\end{equation*}
$$

While in the parallel implementation of the odd-even merge algorithm when run on P processors, the total complexity is equal to,

$$
\begin{align*}
& C_{P M} \leqslant \frac{N M}{P}\left(\frac{M-1}{M-2}\right)-\frac{M^{2}}{2 P}\left(\frac{M-1}{M-2}\right)+2, \text { for } \frac{M}{2}<P,  \tag{6.2.12a}\\
& C_{P M} \leqslant N\left(\frac{2 M P-M-2 P}{2 P(P-1)}\right)-\frac{1}{4} \frac{M^{2}}{P}-\frac{M / 2(M / 2-1)}{(P-1)}+2, \text { for } P=\frac{M}{2}, \tag{6.2.12b}
\end{align*}
$$

and

$$
\begin{equation*}
C_{P M} \leqslant \frac{N}{P}(M-1)-\frac{M^{2}}{2 P}+\frac{M}{2 P}+2, \text { for } \frac{M}{2}>P \tag{6.2.12c}
\end{equation*}
$$

The total complexity of the parallel shell sort algorithm Program 6.4 can be calculated from the complexities of both the sorting procedure (equations (6.2.2) and (6.2.3)) and the odd-even algorithm (equation (6.2.11) and (6.2.12)). Now the total complexity of the Version II parallel shell sort algorithm (i.e. sorting and merging) when run on one processor is,

$$
\begin{equation*}
T_{1}=M\left(\frac{N}{M}\right)^{3 / 2}+N(M-1)-\frac{1}{2} M(M-1) \tag{6.2.13}
\end{equation*}
$$

While the total time complexity $T_{P}$ is obtained from the formula,

$$
\begin{gather*}
T_{P}=T_{P S}+C_{P M} \text { which is equal to, } \\
T_{P} \leqslant\left\{\begin{array}{l}
\left(\frac{M}{P}\right)\left(\frac{N}{M}\right)^{3 / 2}+\frac{N M}{P}\left(\frac{M-1}{M-2}\right)-\frac{M^{2}}{2 P}\left(\frac{M-1}{M-2}\right)+3, \text { for } \frac{M}{2}<P \\
\left(\frac{M}{P}\right)\left(\frac{N}{M}\right)^{3 / 2}+N\left(\frac{2 M P-M-2 P}{2 P(P-1)}\right)-\frac{1}{4} \frac{M^{2}}{P}-\frac{M / 2(M / 2-1)}{(P-1)}+3, \text { for } \frac{M}{2}=P \\
\left(\frac{M}{P}\right)\left(\frac{N}{M}\right)^{3 / 2}+\frac{N}{P}(M-1)-\frac{M^{2}}{2 P}+\frac{M}{2 P}+3, \text { for } \frac{M}{2}>P
\end{array}\right. \tag{6.2.14}
\end{gather*}
$$

Now the merge speed-up, $S_{\text {merge }}{ }^{(M)}$, of $M$ subsets can be calculated by dividing $C_{1 M}$ of equation (6.2.11) by $C_{P M}$ of equation (6.2.12), which gives,
$S_{\text {merge }}(M) \geq\left\{\begin{array}{l}\left(N(M-1)-\frac{1}{2}\left(M^{2}-M\right)\right) /\left(\frac{N M}{P}\left(\frac{M-1}{M-2}\right)-\frac{M^{2}}{2 P}\left(\frac{M-1}{M-2}\right)+2\right), \text { for } \frac{M}{2}<P \\ \left(N(M-1)-\frac{1}{2}\left(M^{2}-M\right)\right) /\left(N\left(\frac{2 M P-M-2 P}{2 P(P-1)}\right)-\frac{M^{2}}{4 P}-\frac{M / 2(M / 2-1)}{(P-1)}+2\right), \text { for } \frac{M}{2}=P \\ \left(N(M-1)-\frac{1}{2}\left(M^{2}-M\right)\right) /\left(\frac{N}{P}(M-1)-\frac{M^{2}}{2 P}+\frac{M}{2 P}+2\right), \text { for } \frac{M}{2}>P\end{array}\right.$
These speed-up formulae can be simplified as an order quantity, such as,

$$
S_{\text {merge }}(M) \geqslant\left\{\begin{array}{l}
O\left(\frac{P(M-2)}{M}\right), \text { for } \frac{M}{2}<P  \tag{6.2.16}\\
O\left(\frac{2 P(P-1)(M-1)}{2 M P-M-2 P}\right), \text { for } \frac{M}{2}=P \\
O(P), \text { for } \frac{M}{2}>P \text {. }
\end{array}\right.
$$

While the total speed-up ( $S_{\text {total }}$ ) is calculated by dividing $T_{1}$ of equation ( 6.2 .13 ) by $T_{p}$ of equation (6.2.14) which gives,

$$
\begin{equation*}
S_{\text {total }} \geqslant\left(M\left(\frac{N}{M}\right)^{3 / 2}+N(M-1)-\frac{1}{2} M(N-1) / T_{P},\right. \tag{6.2.17}
\end{equation*}
$$

where $T_{P}$ takes any value from equation (6.2.14) for the appropriate values of $M$ and $P$.

From the experimental results obtained from the implementation of the parallel shell sort algorithm using the parallel odd-even merge algorithm (Table 6.6), we notice that the sorting time decreases as the number of subgroups (M) increase and this is due to the overheads for contention of the parallel paths. Also we notice that the sorting speed-up obtained when the number of subgroups is equal to the number of available processors ( $M=P$ ). While the time for merging increases as the number of subgroups increases and this is due to the fact that the number of comparisons increases with the number of subgroups. The best merging speed-up is obtained when $M=128$. On the other hand, generally the total time increases as the number of subgroups increase and the speed-up is of $O(P)$ which increases as the number of subgroups increases, while the best speed-up occurs when $M=128$. Table 6.7 also shows the experimental results with the total efficiency ( $E_{P}$ ) of this algorithm, where the best total time is obtained when $M$ is equal to 4 in the case of $P$ equal to 1 and 4 and $M=8$, for the case $P=2$ and 3 . The best efficiency value obtained using different processors is when $M=128$ and the efficiency decreases when the number of processors increases. This is due to the reduction in the usage of the processors in each step of the algorithm.

To compare the results obtained from Version II of the parallel
shell sort (i.e. the implementation that using the parallel 2-way merge and the parallel odd-even merge), we notice that the time required for the input data to be sorted using the parallel 2-way merge is less than that using the parallel odd-even merge. This is because the number of steps in the 2 -way merge is log $M$ while in the odd-even merge it is $M$ steps which is very much greater than $\operatorname{logM}$ and the number of parallel paths in the case of odd-even merge is greater than that of the 2 -way merge as shown in Tables 6.4 and 6.7 , which means more time is required for the algorithm to sort the input data when it uses the parallel odd-even merge algorithm. The total speed-up of the algorithm using the odd-even merge algorithm is generally higher than that of using the 2 -way merge and this is due to the higher speedup in the merge phase which affects the total speed-up. Thus, for parallel implementation if we consider the complexity of both methods, tr.: run-time illustrates that the shell sort with the 2 -way merge method for a particular data size is better than that using the odd-even merge method. Also from Tables 6.3 and 6.6 we notice that the odd-even merge speed-up is better than the 2 -way merge speed-up. This is mainly due to the usage of processors where in the odd-even merge the processors were mostly fully used in all the steps of the merge phase while in the 2 -way merge the number of processors is halved in each step.

For comparison reasons, Table 6.8 shows the experimental results obtained from the implementation of the parallel shell sort methods on the NEPTUNE system using Version I and II with data size 1024. We notice that the best timing results are obtained when the sorting is carried out by using version II with the 2 -way merge method, with the Version II methods giving better speed-up results than version I methods.

In general, the speed-up of all the implemented methods are not high and this is due to the fact that the processors are not fully utilized in the different sorting steps and specially when more than 2 processors are used. The timing results in Table 6.8 are graphically represented in Figure 6.4 which shows that Version II with the 2 -way merge is the best. For a performance analysis using the facilities of the NEPTUNE system and with reference to Chapter 4 , the parallel control overheads (PCO) and shared data overheads (SDO) of the different implemented parallel shell methods are tabulated in Table 6.9. For Version II, the timing results used are the ones that give the best efficiency, i.e. for the 2-way merge the results used are for $M=64$ ( $M$ equal to the number of subgroups), while for the odd-even merge the one used is for M=128, which gives the best efficiency. From Table 6.9 it is clear that for Version $I$, the amount of the overheads using the second approach is less than that of using the first approach and this is due to the fact that the number of parallel paths generated in the second approach is less than the first approach, as shown in Tables 6.1 and 6.2. While for Version II, the algorithm using the 2 -way merge gives better results than that using the odd-even merge and is due to the number of generated parallel paths using the 2-way merge method being less than that used by the odd-even merge as shown in Tables 6.4 and 6.7. The results in Table 6.9 show that the parallel shell sort method using the 2 -way merge algorithm gives the best results since it needs the lowest overheads.

### 6.2.2 The Digit Sort Method

In the previous section the implemented shell method belongs to

| No. Of Processors (P) | No. $0 f$ <br> Paths <br> (M) | Time for Sorting (sec.) | Time for Merging (sec.) | Total <br> Time <br> (sec.) | ```Speed-up for Sorting``` | ```Speed-up for Merging``` | Speed-up <br> for Both <br> Parts |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 2 | 9.10 | 0.85 | 9.95 | 1.0 | 1.0 | 1.0 |
| 2 | 2 | 4.62 | 0.85 | 5.47 | 1.96970 | 1.0 | 1.81901 |
| 1 | 4 | 7.45 | 2.33 | 9.78 | 1.0 | 1.0 | 1.0 |
| 2 | 4 | 3.79 | 1.72 | 5.51 | 1.96570 | 1.35465 | 1.77495 |
| 4 | 4 | 1.94 | 1.72 | 3.66 | 3.84021 | 1.35465 | 2.54674 |
| 1 | 8 | 6.16 | 4.57 | 10.73 | 1.0 | 1.0 | 1.0 |
| 2 | 8 | 3.14 | 2.87 | 6.01 | 1.96178 | 1.5923 | 1.78536 |
| 3 | 8 | 2.2 | 2.25 | 4.45 | 2.80000 | 2.03111 | 2.41124 |
| 4 | 8 | 1.61 | 1.88 | 3.49 | 3.82609 | 2.43085 | 3.07450 |
| 1 | 16 | 4.72 | 8.89 | 13.61 | 1.0 | 1.0 | 1.0 |
| 2 | 16 | 2.41 | 5.09 | 7.50 | 1.95851 | 1.74656 | 1.81467 |
| 3 | 16 | 1.69 | 3.92 | 5.61 | 2.79290 | 2.26786 | 2.42602 |
| 4 | 16 | 1.24 | 3.15 | 4.39 | 3.80645 | 2.82222 | 3.10023 |
| 1 | 32 | 3.61 | 17.7 | 21.31 | 1.0 | 1.0 | 1.0 |
| 2 | 32 | 1.85 | 9.53 | 11.38 | 1.95135 | 1.85729 | 1.87258 |
| 3 | 32 | 1.30 | 6.83 | 8.13 | 2.77692 | 2.59151 | 2.62116 |
| 4 | 32 | 0.95 | 5.43 | 6.38 | 3.80000 | 3.25967 | 3.34013 |
| 1 | 64 | 2.62 | 35.88 | 38.50 | 1.0 | 1.0 | 1.0 |
| 2 | 64 | 1.35 | 18.74 | 20.09 | 1.94074 | 1.91462 | 1.91638 |
| 3 | 64 | 0.95 | 12.98 | 13.93 | 2.75790 | 2.76425 | 2.76382 |
| 4 | 64 | 0.71 | 10.20 | 10.91 | 3.69014 | 3.51765 | 3.52887 |
| 1 | 128 | 1.82 | 74.65 | 76.47 | 1.0 | 1.0 | 1.0 |
| 2 | 128 | 0.94 | 38.45 | 39.39 | 1.93617 | 1.94148 | 1.94136 |
| 3 | 128 | 0.66 | 26.48 | 27.14 | 2.63768 | 2.81911 | 2.81761 |
| 4 | 128 | 0.50 | 20.48 | 20.98 | 3.64000 | 3.64502 | 3.64490 |
| 1 | 256 | 1.25 | 163.17 | 164.42 | 1.0 | 1.0 | 1.0 |
| 2 | 256 | 0.65 | 84.11 | 84.76 | 1.92308 | 1.93996 | 1.93983 |
| 3 | 256 | 0.48 | 57.99 | 58.47 | 2.60417 | 2.81376 | 2.81204 |
| 4 | 256 | 0.35 | 45.5 | 45.85 | 3.57143 | 3.58615 | 3.58604 |

TABLE 6.6: The experimental results of the shell sort algorithm using the odd-even merge algorithm (Version II) for input size 1024.

| $\begin{aligned} & \text { Size } \\ & \mathrm{N} \end{aligned}$ | Processors (P) | Parallel Paths <br> (M) | Total <br> Time (sec.) | Total Speed-up | Total <br> Parallel <br> Paths for <br> Sorting | Total <br> Parallel <br> Paths for Merging | Total <br> Effici- <br> ency <br> ( $\mathrm{E}_{\mathrm{p}}$ ) |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1024 | 1 | 2 4 8 16 32 64 128 256 | $\begin{array}{r} 9.95 \\ 0.78 \\ 10.73 \\ 13.61 \\ 21.31 \\ 38.50 \\ 76.47 \\ 164.42 \end{array}$ | $\begin{aligned} & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \end{aligned}$ | $\begin{array}{r} 5 \\ 7 \\ 11 \\ 19 \\ 35 \\ 67 \\ 131 \\ 259 \end{array}$ | $\begin{array}{r} 4 \\ 12 \\ 38 \\ 138 \\ 530 \\ 2082 \\ 8258 \\ 32898 \end{array}$ | $\begin{aligned} & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \\ & 1.0 \end{aligned}$ |
|  | 2 | $\begin{array}{r} 2 \\ 4 \\ 8 \\ 16 \\ 32 \\ 64 \\ 128 \\ 256 \end{array}$ | $\begin{array}{r} 5.47 \\ 5.51 \\ 6.01 \\ 7.50 \\ 11.38 \\ 20.09 \\ 39.39 \\ 84.76 \end{array}$ | $\begin{aligned} & 1.81901 \\ & 1.77495 \\ & 1.78536 \\ & 1.81467 \\ & 1.87258 \\ & 1.91638 \\ & 1.94136 \\ & 1.93983 \end{aligned}$ | $\begin{array}{\|c} 4,2 \\ 5,3 \\ 7,5 \\ 11,9 \\ 19,17 \\ 35,33 \\ 67,65 \\ 133,127 \end{array}$ | 4,1 10,3 24,15 79,60 283,248 1085,998 4217,4042 16683, 16216 | $\begin{aligned} & 0.91 \\ & 0.89 \\ & 0.89 \\ & 0.91 \\ & 0.94 \\ & 0.96 \\ & 0.97 \\ & 0.97 \end{aligned}$ |
|  | 3 | 8 <br> 16 <br> 32 <br> 64 <br> 128 <br> 256 | $\begin{array}{r} 4.45 \\ 5.61 \\ 8.13 \\ 13.93 \\ 27.14 \\ 58.47 \end{array}$ | $\begin{aligned} & 2.41124 \\ & 2.42602 \\ & 2.62116 \\ & 2.76382 \\ & 2.81761 \\ & 2.81204 \end{aligned}$ | $\begin{gathered} 6,4,3 \\ 9,6,6 \\ 14,11, \\ 12 \\ 25,22, \\ 22 \\ 47,43, \\ 43 \\ 89,85 \\ 87 \\ \hline \end{gathered}$ | $\begin{gathered} 19,10,11 \\ 59,40,41 \\ 202,164, \\ 166 \\ 762,665, \\ 657 \\ 2898,2679, \\ 2683 \\ 11351,10794, \\ 10755 \end{gathered}$ | $\begin{aligned} & 0.80 \\ & 0.81 \\ & 0.87 \\ & 0.92 \\ & 0.94 \\ & 0.94 \end{aligned}$ |
|  | 4 |  | $\begin{array}{r} 3.66 \\ 3.49 \\ 4.39 \\ 6.38 \\ 10.91 \\ 20.98 \\ 45.85 \end{array}$ | 2.54674 3.07450 3.10023 3.34013 3.52887 3.64490 3.58604 | $\begin{gathered} 4,2,2,2 \\ 5,3,3,3 \\ 7,5,5,5 \\ 11,9,9,9 \\ 19,17, \\ 17,17 \\ 36,32, \\ 33,33 \\ 70,66, \\ 66,65 \end{gathered}$ | $\begin{gathered} 10,1,2,2 \\ 18,8,9,6 \\ 48,32,32, \\ 29 \\ 159,125, \\ 124,125 \\ 577,499, \\ 499,510 \\ 2219,2015, \\ 2009,2018 \\ 8657,8077, \\ 8079,8088 \end{gathered}$ | $\begin{aligned} & 0.64 \\ & 0.77 \\ & 0.78 \\ & 0.84 \\ & 0.88 \\ & 0.91 \\ & 0.90 \end{aligned}$ |

TABLE 6.7: The experimental results of the total efficiency of the parallel shell sort (Version II with odd-even merge)

| Program | Processors |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 1 | 2 |  | 3 |  | 4 |  |
|  | Time | Time | Speed-up | Time | Speed-up | Time | Speed-up |
| Version I lst strategy | 22.41 | 15.88 | 1.41121 | 14.31 | 1.56604 | 13.60 | 1.64779 |
| Version I 2nd strategy | 11.18 | 6.72 | 1.66369 | 4.85 | 2.30516 | 4.32 | 2.58796 |
| Version II with 2 -way merge | 7.49 | 4.28 | 1.75000 | 3.52 | 2.12784 | 2.86 | 2.61888 |
| Version II with oddeven merge | 9.78 | 5.47 | 1.78793 | 4.45 | 2.19775 | 3.66 | 2.67213 |

TABLE 6.8: The experimental results obtained from the implementation of parallel shell sort with data size 1024.

| Program | $T_{p}$ | $T_{N}$ | $T_{S}$ | PCO | SDO |
| :---: | :---: | :---: | :---: | :--- | :--- |
| Version I, lst <br> Approach | 22.41 | 21.7 | 21.4 | $3.17 \%$ | $1.34 \%$ |
| Version I, 2nd <br> Approach | 11.18 | 10.85 | 10.72 | $2.95 \%$ | $1.16 \%$ |
| Version II,2-way | 2.62 | 2.57 | 2.56 | $1.91 \%$ | $0.38 \%$ |
| Sort | 4.87 | 4.83 | 4.79 | $0.82 \%$ | $0.82 \%$ |
| Merge | 7.49 | 7.40 | 7.35 | $1.2 \%$ | $0.67 \%$ |
| Total |  |  |  |  |  |
| Version II,Odđ-Even | 1.82 | 1.75 | 1.73 | $3.85 \%$ | $1.1 \%$ |
| Sort | 74.65 | 67.44 | 66.71 | $9.66 \%$ | $0.98 \%$ |
| Merge | 76.47 | 69.16 | 68.42 | $9.56 \%$ | $0.97 \%$ |

TABLE 6.9: The performance results of the parallel shell sort


FIGURE 6.4

The experimental total timing results obtained from the implementation of the parallel shell sort methods using data size equal to 1024 .
a class of sorting methods known as the comparative methods in which the data is ordered and depends on the comparison of the relative magnitude of keys on the list. In this section we will implement another sorting method which belongs to an alternative class known as the distributive method, where each key inspects either character by character or as an entity. By their nature distributive sorts are not minimal-storage techniques. Since the "distribute" elements receive areas on the basis of some characteristic of the key, there must be an allocation of space for receiving such areas other than the space used by their initial list. The digit sort algorithm belongs to the distributive class and the sort is carried out by distributing the elements into receiving areas based on the value of a specific digit of the key.

The method consists of partially sorting the records by a digit sort and completing the sort by the bubble sort method. A file of records $R_{1}, R_{2}, \ldots, R_{n}$ is to be sorted according to the rank of the keys $x_{1}, x_{2}, x_{3}, \ldots, x_{n}$. The keys are represented as numbers of $d$ digits in the number system of base $m$. The base $m$ may be chosen arbitrarily. The proposed method, may be thought of as consisting of three steps.

The first step is an adaptation of the familiar digit sort, which is used to sort punch cards on a card sorter. In this step, the records are sorted only on the $P$ high digits of the keys. The digit sort is carried out in $P$ passes and at the end of the pth pass, the records are sorted according to the $P$ high order digits of the keys.

The second step in the method is to complete the sorting by the familiar bubble method. The advantage of the bubble sort method is that it takes very little time when the records are almost in order, as
they will be after the first step. The final step is to use the list information to write the records out into an external memory or into another area in the main memory or to rearrange them in place. It happens that, unless the records are being written into an external memory, the second and the third steps can be effectively combined. The feature which makes the digit sort attractive is that the time required per record to sort $n$ records is independent of $n$ (Maclaren [1966]).

As a special case of the application of this method we consider the rearrangement of an array $\left\{X_{i}\right\}$ of positive integers so that $X_{1} \leqslant X_{2} \leqslant$ $\ldots \leqslant X_{n}$. It is supposed that the number $X_{i}$ are represented to the base m. The highest order digit of $X_{i}$ is denoted by $F_{p}\left(X_{i}\right)$, the second highest by $F_{p-1}\left(X_{i}\right)$, etc. Here the $X_{i}$ are assumed to be the number of $d$ digits and the high order digits may be zero. The integer $p$ is the number of passes to be made in the digit sort. $P$ can be any integer not exceeding $d$ and $P=2$, is apparently the best choice (Maclaren [1966]). The general idea in the digit sort is to have an array of $n$ lists (which are called bins), numbered $0,1, \ldots, m-1$. On pass number $K$, the record $X_{i}$ is put in bin number $F_{k}\left(X_{i}\right)$. At the end of the pass all the bins are, in effect, put together to form one list containing all the $X_{i}$. After $P$ passes of the digit sort have been completed the numbers have been sorted on their $P$ highest order digits. For $j<k<m^{p}$ every number falling in the interval $\left[j m^{d-p},(j+1) m^{d-p}\right]$ will precede every number in the interval $\left[k m^{d-p},(k+1) m^{d-p}\right]$, again considering all the bins as combined into one list. If the total number of intervals $\mathrm{m}^{\mathrm{p}}$ is suitably large compared with n , the expected number of $\mathrm{X}_{\mathrm{i}}$ falling in any one interval will not be large. This suggests completing the
sort by a procedure such as the bubble sort, which uses only few operations when the records are almost in order to begin with.

The bubble sort procedure can be described as, to sort $n$ numbers $X_{1}, x_{2}, \ldots, x_{n}, n$ steps may be required. At the end of the ( $j-1$ ) th steps, the first $(j-1)$ numbers have been rearranged so that $X_{1} \leqslant X_{2} \leqslant \ldots \leqslant X_{j-1}$. On step number $j, X_{j}$ is compared with $X_{j-1}$; then if necessary, $X_{j-2}$, $X_{j-3}$, etc., until $X_{i}$ is found so that $X_{j} \geqslant X_{i}$. At that point $X_{i}$ is found so that $X_{j} \geqslant X_{i}$ and $X_{j}$ is inserted between $X_{i}$ and $X_{i+1}$. If the set of numbers to begin with is in order this procedure uses only $n$ comparisons otherwise it is time consuming.

## The Parallel Digit Sort

Two parallel versions of the sequential digit sort have been implemented on the NEPTUNE system and these implementations are programmed in Program (6.5) and (6.6) respectively. The three stages of the digit sort are shown in Figure 6.5 and the first parallel implementations are carried out as follows:
(1) Sl: The $n$ keys are presorted into $m$ "bins" according to their first $P$ digits and transferred sequentially to the available processors.
(2) Pl: The processors sort internally their bins into runs using the bubble sort procedure.
(3) S2: The runs are sent sequentially into the common memory where
they constitute the sorted list (transfer of runs $1,2, \ldots, m$ ).


FIGURE 6.5: The stages of first parallel digit sort implementation

The digit sorting of the first step is based on the assumption that there exists an integer $P$ such that the list may be sorted into $m$ bins by converting the keys to base $m$ and taking the first $p$ digits as indicating the bin to which the key belongs. In our implementation, $P$ is taken to be equal to 2 and the bins will include approximately $\mathrm{n} / \mathrm{m}$ keys (Maclaren [1966]). As an example, Table 6.10 shows a list of numbers to be sorted, first we convert the numbers to base $m=5$. Lists are formed according to whether the first digit is $0,1,2,3$ or 4 and sublists according to the second digit as shown in Table 6.11.

```
316,3736,5477,29968,16533,4136,3813,32758,11584,
21735,22034,13745,25025,4770,10131,4871,12588,23182,19270,413
```

TABLE 6.10: The list of N items to be sorted

| $\begin{aligned} & \text { Base } 10 \\ & \text { Base } 5 \end{aligned}$ | BIN 00 |  | BIN Ol |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 316 | 413 | 3736 | 5477 | 4136 | 3813 | 4770 | 4871 |
|  | $0002231 \quad 0003123$ |  | 0104421 | 0133404 | 0113021 | 0110223 | 0123041 | 0123441 |
|  | BIN 03 |  | BIN 04 |  | BIN 10 | BIN 11 |  |  |
|  | 1158410131 |  | 1374512588 |  | 16533 | 2173519270 |  |  |
|  | $0332314 \quad 03311011$ |  | $0414440 \quad 0400323$ |  | 1011113 | $1143420 \quad 1104040$ |  |  |
|  | BIN | 12 |  | 13 | BIN 14 | BIN |  |  |
|  | 22034 | 23182 | 250 |  | 29968 | 3275 |  |  |
|  | 1201114 | 1220212 | 130 | 100 | 1424333 | 2022 |  |  |

TABLE 6.11: The list and sublists of $N$ items to be sorted
Bins 02, 21, 22, 23, 24 are empty and no main lists starts with 3 or 4 . The number of elements included in the lists and sublists may be shown in a "matrix" $A$ of $P$ dimensions (Table 6.12).


TABLE 6.12: Matrix A of list

For $P=2$, the columns of Table 6.12 represent the sublists, the rows the main lists. For example, element $A(2,3)=2$ shows that there are 2 elements with leading element 1 and sublist 2, i.e. in bin 12 (see Table 6.11). Matrix $A$ is helpful in dividing the list into approximately equally long bins (of $\mathrm{n} / \mathrm{m}$ records). The number of operations required for deciding where to send the various lists and sublists is a function of $\mathrm{m}^{\mathrm{p}}$. Since there is no connection between m and $n$, the highest speedup is expected for very large $n$.

The flowchart of the first parallel digit sort algorithm using general $P$ is shown overleaf (Program 6.5).


In Version $I$, the distribution is done by one processor only. For instance, the processing of key $=23183$ shows the following steps:
$K(1)=\frac{23182}{15625}=1$; insert 23182 into main 1ist 1 ,
$R=23182-1 * 15625=7557$; $K(2)=\frac{7557}{3125}=2$; insert 23182
into sublist 12 ; update element $A(1,2)$ by adding 1 to it.
Version II of the parallel digit sort is implemented on the NEPTUNE system in the same way as Version I except that the distribution of records (step $S l$ in Figure 6.5) into main and sublists are done in parallel instead of sequential (one processor), i.e. the input (N) is subdivided into subgroups each with (N/NPATH) elements, where NPATH is equal to the number of subgroups, and assign one group to each of the available processors, in which its elements will be distributed over sublists and matrix A update accordingly. In order to maintain the consistency of the information in this implementation a critical section is used while updating the matrix $A$ and updating the links between the main and sublists. The usage of critical section will affect the performance of the algorithm as seen later.

The experimental results obtained from the implementation of the Version I parallel digit sort on the NEPTUNE system are shown in Tables 6.13 and 6.14. The results in Table 6.13 are obtained when the data size sorted is equal to 1024 , while Table 6.14 represents the results obtained when the data size sorted is equal to 2048. In Table 6.13 different timing results obtained for the different number system bases (i.e. which convert the unsorted numbers to a specific number system base), and it can be noticed that the time required for the sorting is proportional to the number of subsets obtained, where higher sorted time required when the unsorted input set is divided into a small number of
subsets and vice versa. In our implementation the best time obtained (lowest time) when the number system base used is equal to 8 and this is due to the fact that this number system gives the highest number of subsets amongst the other implemented number system bases (which is equal to 64 subsets). For the higher number of subsets means a small number of elements in each subset and as we know that the implemented sorting procedure is efficient for a list of almost sorted small number of elements. Also it can be noticed that for the sort phase only the best efficiency (speed-up) is obtained when we get a higher number of subsets and this is due to the fact that the processors are fully utilized and a better efficiency obtained. For the total speed-up (i.e. for both sorting and distribution phases) we notice that it is not as good as the sorting phase only and this is due to the way in which the algorithm is implemented. In this implementation the distribution part is carried out sequentially, i.e. by one processor only and this will increase the total timing and decrease the total speed-up of the algorithm. Because the distribution time is the same for all the different number systems a high effectiveness will occur in the case when the number of the subsets is equal to 64 (base 8), i.e. in the case of the lowest time as shown in Table 6.13 where the efficiency of the processors dropped from 1.97698 to $1.77173,2.96169$ to 2.40274 and 3.94388 to 2.92333 for 2,3 and 4 processors respectively. Therefore, for the total sorting time the best efficiency was achieved when the number of subsets is a multiple of the number of processors, where for 2 and 4 processors the best efficiency is obtained when the number of subsets is equal to 4 and for 3 processors the number of subsets is equal to 26 gives the best efficiency. For the results in Table 6.14
i.e. the input data size equal to 2048 ), we notice that the same observations are obtained as that of input data size equal to 1024 with the exception that the best total speed-up obtained for 3 and 4 processors when number of subsets is equal to 14 and 8 respectively.

By comparing the results in Tables 6.13 and 6.14 we notice that a higher efficiency (speed-up factors) is obtained in the case of a higher input data aize (i.e. the speed-up results for the data size 2048 is better than that of data size 1024) and this agrees with the theoretical results (Maclaren [1966]). Generally, we can say that better speed-up results will be expected with a higher input data size.

The experimental results of the version II parallel digit sort using input data size 1024 and 2048 are shown in Tables 6.15 and 6.16 respectively. From these two tables it is clear that the total sorting time is decreased as the number of subsets is increased. This is due to the fact that the sorting procedure is efficient when the number of elements in each subset are small and almost sorted and this is what we get with the higher number of subsets. As in Version $I$, the best timing is obtained (the lowest sort time) when the number of subsets is equal to 64 which is obtained when the number system base is equal to 8 . For the sorting part with input size 1024, the best speed-up (efficiency) is achieved when the number of subsets is equal to 26 when using 2 processors and 64 subsets when using 3 and 4 processors. While for the total sorting time the best efficiency is obtained when the number of subsets is equal to 26 when using 2 and 3 processors and 4 subsets when using 2 processors. From the results in Table 6.16 where the input data size is 2048, we notice that for the sorting part only the best speed-up factors are obtained when number of subsets is equal to 64,26 and 64
\(\left.$$
\begin{array}{|l|l|l|l|l|l|l|l|}\hline \begin{array}{l}\text { No. of } \\
\text { Processors }\end{array} & \text { Base } & \begin{array}{l}\text { No.of } \\
\text { Subset }\end{array} & \begin{array}{l}\text { Sort } \\
\text { (sec.) }\end{array} & \begin{array}{c}\text { Distribution } \\
\text { (sec.) }\end{array} & \begin{array}{l}\text { Total } \\
\text { Time } \\
\text { (sec.) }\end{array} & \begin{array}{l}\text { Speed-up } \\
\text { of } \\
\text { Sorting }\end{array} & \begin{array}{l}\text { Total } \\
\text { Speed- } \\
\text { up }\end{array}
$$ <br>
\hline 1 \& 2 \& 4 \& 97.31 \& 1.04 \& 98.35 \& 1.0 \& 1.0 <br>
2 \& \& \& 50.77 \& 1.04 \& 51.81 \& 1.91668 \& 1.89828 <br>
3 \& \& \& 48.81 \& 1.04 \& 49.85 \& 1.99365 \& 1.97292 <br>

4 \& \& \& \& 25.61 \& 1.04 \& 26.65 \& 3.79969\end{array}\right] 3.69043\)| 1.04 |
| :--- |
| 1 |
| 2 |

TABLE 6.13: The timing results obtained from the implementation of the Version $I$ parallel digit sort using data size equal to 1024.

| No. of Processors | Base | No. 0 f <br> Subset | Sort (sec.) | Distribution (sec.) | Total Time (sec.) | Speed-up of Sorting | Total <br> Speedup |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 2 | 4 | 391.55 | 2.03 | 393.58 | 1.0 | 1.0 |
| 2 |  |  | 201.13 | 2.03 | 203.16 | 1.94675 | 1.93729 |
| 3 |  |  | 189.49 | 2.03 | 191.52 | 2.06634 | 2.05503 |
| 4 |  |  | 103.01 | 2.03 | 105.04 | 3.80109 | 3.74695 |
| 1 | 3 | 5 | 312.40 | 2.03 | 314.43 | 1.0 | 1.0 |
| 2 |  |  | 182.93 | 2.03 | 184.96 | 1.70776 | 1.69999 |
| 3 | . |  | 126.12 | 2.03 | 128.15 | 2.47701 | 2.45361 |
| 4 |  |  | 121.09 | 2.03 | 123.12 | 2.57990 | 2.55385 |
| 1 | 4 | 8 | 192.86 | 2.03 | 194.89 | 1.0 | 1.0 |
| 2 |  |  | 98.73 | 2.03 | 100.76 | i. 95341 | 1.93420 |
| 3 |  |  | 74.75 | 2.03 | 76.78 | 2.58007 | 2.53829 |
| 4 |  |  | 49.78 | 2.03 | 51.81 | 3.87425 | 3.76163 |
| 1 | 5 | 11 | 143.06 | 2.03 | 145.09 | 1.0 | 1.0 |
| 2 |  |  | 73.20 | 2.03 | 75.23 | 1.95437 | 1.92862 |
| 3 |  |  | 54.90 | 2.03 | 56.93 | 2.60583 | 2.54857 |
| 4 |  |  | 40.54 | 2.03 | 42.57 | 3.52886 | 3.40827 |
| 1 | 6 | 26 | 60.40 | 2.03 | 62.43 | 1.0 | 1.0 |
| 2 |  |  | 30.53 | 2.03 | 32.56 | 1.97838 | 1.91738 |
| 3 |  |  | 20.55 | 2.03 | 22.58 | 2.93917 | 2.76484 |
| 4 |  |  | 16.39 | 2.03 | 18.42 | 3.68517 | 3.38925 |
| 1 | 7 | 14 | 111.19 | 2.03 | 113.22 | 1.0 | 1.0 |
| 2 |  |  | 58.40 | 2.03 | 60.43 | 1.90394 | 1.87357 |
| 3 |  |  | 38.54 | 2.03 | 40.57 | 2.88505 | 2.79073 |
| 4 |  |  | 32.85 | 2.03 | 34.88 | 3.38478 | 3.44658 |
| 1 | 8 | 64 | 27.29 | 2.03 | 29.32 | 1.0 | 1.0 |
| 2 |  |  | 13.72 | 2.03 | 15.75 | 1.98907 | 1.86159 |
| 3 |  |  | 9.20 | 2.03 | 11.23 | 2.96630 | 2.61086 |
| 4 |  |  | 6.90 | 2.03 | 8.93 | 3.95507 | 3.28700 |

TABLE 6.14: The timing results obtained from the implementation of the Version I parallel digit sort using data size equal to 2048.
when using 2,3 and 4 processors respectively. While for the total sorting time the best efficiency (speed-up) is achieved when the number of subsets is equal to 26,64 and 8 when using 2,3 and 4 processors respectively. It is noticed that the efficiency (speed-up) of the distribution part is not as good as that of the sorting part and this will affect the overall efficiency of the algorithm. The reason for this is the update of matrix "A" and the links between the lists and sublists are performed within a critical section (by using the \$ENTER and \$EXIT constructs). This means only one processor can carry out the updating while the rest of the processors are idle, which greatly affects the performance of the algorithm. From the experimental results in Tables 6.15 and 6.16 we also notice that a better efficiency (speedup) is achieved for a higher input data size, i.e. the speed-up obtained from using data size 2048 (Table 6.16) is better than that of using data sịze 1024 (Table 6.1) and this is confirmed by the results in parallel Version I.

In Version II the usage of the critical section in the distribution part has a significant delay that causes a higher running time and a low speed-up is obtained since the overheads for the critical section accesses are much more than the parallelism gain in the algorithm. In order to make clear the amount of degradation that is caused by the usage of the critical section we need to know the time spent to access the critical section 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, in the NEPTUNE system the time required to access a critical section (i.e. the \$ENTER/\$EXIT construct) is $\sim 800 \mu s e c o n d s$ and the cycle time while waiting to enter a critical
section is $1080 \mu \mathrm{sec}$ ands. In our implementation there is one access to the critical section for each element in the input data set. Therefore to sort $N$ elements the algorithm needs $N$ accesses to the critical section. Besides the time spent in the critical section, there is a waiting time by the processor to enter a critical section. Thus, the time spent in the critical section $=$ number of accesses to the section $\times 800 \mu s$,
and
the time spent for waiting cycles to access critical section $=$ number of waiting cycles $\times 1080 \mu \mathrm{~s}$.

For Version II of the parallel digit sort algorithm the total times caused by the usage of critical section are shown in Table 6.17. It is noticed that the time lost while using an input data size 2048 is greater than that from using 1024. This is due to the fact that for each element in the input data set one access to the critical section is needed. Besides the critical section time lost, Table 6.17 shows the performance analysis of the parallel digit sort algorithms with both the shared data (SDO) and parallel control overheads (PCO) are calculated (see Chapter 4). From the results in Table 6.17 we can notice that the speed-up factors of the parallel version II method is better than that of the parallel version $I$ method and the amount of overheads in the parallel version II method is less than that of the parallel Version I method. So we can conclude that generally the parallel digit method Version II is more suitable for the MIMD parallel type machine than that of Version $I$.



| Base | Subsets | No.of <br> Paths | No.of <br> Processors | Sorting (sec.) | Distribution (sec.) | $\begin{aligned} & \text { Total Time } \\ & \text { (sec.) } \end{aligned}$ | Speed-up for Sorting | Speed-up <br> for Distribution | Total Speed-up |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 8 | 64 | $\begin{aligned} & 1 \\ & 2 \end{aligned}$ | 1 | 7.71 | 1.83 | 9.54 |  |  |  |
|  |  |  | 1 | 7.70 | 1.83 | 9.53 |  |  |  |
|  |  |  | 2 | 3.91 | 0.97 | 4.88 | 1.96931 | 1.88660 | 1.95287 |
|  |  | 3 | 1 | 7.72 | 1.84 | 9.56 |  |  |  |
|  |  |  | 2 | 3.93 | 1.24 | 5.17 | 1.96438 | 1.48387 | 1.84913 |
|  |  |  | 3 | 2.63 | 1.00 | 3.63 | 2.93536 | 1.84000 | 2.63361 |
|  |  | 4 | 1 | 7.74 | 1.84 | 9.58 |  |  |  |
|  |  |  | 2 | 3.92 | 0.95 | 4.87 | 1.97449 | 1.93684 | 1.96715 |
|  |  |  | 3 | 2.63 | 1.20 | 3.83 | 2.94297 | 1.53333 | 2.50131 |
|  |  |  | 4 | 1.98 | 0.92 | 2.90 | 3.90909 | 2.00000 | 3.30345 |

TABLE 6.15(C): The timing results obtained from the implementation of the Version II parallel digit sort using data aize equal to 1024.


| Base | Subsets | No. of <br> Paths | No. of <br> Processors | Sorting <br> (sec.) | Distribution (sec.) | ```Total Time (sec.)``` | Speed-up for Sorting | ```Speed-up for Distribution``` | Total Speed-up |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 5 | 11 | $\begin{aligned} & 1 \\ & 2 \end{aligned}$ | 1 <br> 1 <br> 2 <br> 1 <br> 2 <br> 3 <br> 1 <br> 2 <br> 3 <br> 4 | 143.33 | 3.18 | 146.51 |  |  |  |
|  |  |  |  | 143.08 | 3.18 | 146.26 |  |  |  |
|  |  |  |  | 73.46 | 1.81 | 75.27 | 1.94773 | 1.75691 | 1.94314 |
|  |  | 3 |  | 142.93 | 3.18 | 146.11 |  |  |  |
|  |  |  |  | 73.37 | 2.36 | 75.73 | 1.94807 | 1.34746 | 1.92935 |
|  |  |  |  | 54.96 | 2.01 | 56.97 | 2.60062 | 1.58209 | 2.56468 |
|  |  | 4 |  | 142.92 | 3.18 | 146.10 |  |  |  |
|  |  |  |  | 73.39 | 1.91 | 75.30 | 1.94740 | 1.63918 | 1.94024 |
|  |  |  |  | 54.95 | 2.31 | 57.26 | 2.60091 | 1.37662 | 2.55152 |
|  |  |  |  | 40.56 | 1.74 | 42.30 | 3.52367 | 1.82759 | 3.45390 |
| 6 | 26 | 12 | 1 | 60.59 | 3.15 | 63.74 |  |  |  |
|  |  |  | 1 | 60.64 | 3.17 | 63.81 |  |  |  |
|  |  |  | 2 | 30.47 | 1.93 | 32.40 | 1.99015 | 1.64249 | 1.96944 |
|  |  | 3 | 1 | 60.62 | 3.17 | 63.79 |  |  |  |
|  |  |  | 2 | 30.5 | 2.32 | 32.82 | 1.98754 | 1.36638 | 1.94363 |
|  |  |  | 3 | 20.56 | 2.00 | 22.56 | 2.94844 | 1.58500 | 2.82757 |
|  |  | 4 | 1 | 60.59 | 3.17 | 63.76 |  |  |  |
|  |  |  | 2 | 30.40 | 1.89 | 32.29 | 1.99309 | 1.67725 | 1.97461 |
|  |  |  | 3 | 20.53 | 2.70 | 23.23 | 2.95129 | 1.17407 | 2.74473 |
|  |  |  | 4 | 16.41 | 1.79 | 18.20 | 3.69226 | 1.77095 | 3.50330 |
| 7 | 14 | 1 | 1 | 111.40 | 3.18 | 114.58 |  |  |  |
|  |  |  | 1 | 111.46 | 3.18 | 114.64 |  |  |  |
|  |  |  | 2 | 58.45 | 1.93 | 60.38 | 1.90693 | 1.64767 | 1.89864 |
|  |  | 3 | 1 | 111.09 | 3.17 | 114.26 |  |  |  |
|  |  |  | 2 | 58.41 | 2.34 | 60.75 | 1.90190 | 1.35470 | 1.88082 |
|  |  |  | 3 | 38.54 | 2.01 | 40.55 | 2.88246 | 1.57711 | 2.81776 |
|  |  | 4 | 1 | 111.07 | 3.17 | 114.24 |  |  |  |
|  |  |  | 2 | 58.40 | 1.87 | 60.27 | 1.90188 | 1.69519 | 1.89547 |
|  |  |  | 3 | 38.55 | 2.29 | 40.84 | 2.88119 | 1.38428 | 2.79726 |
|  |  |  | 4 | 32.47 | 1.76 | 34.23 | 3.42070 | 1.80114 | 3.33742 |


| Base | Subsets | No. of Paths | No. of <br> Processors | Sorting (sec.) | Distribution (sec.) | $\begin{aligned} & \text { Total Time } \\ & \text { (sec.) } \end{aligned}$ | Speed-up <br> for Sorting | Speed-up <br> for Distribution | Total <br> Speed-up |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 8 | 64 | 1 | 1 | 27.20 | 3.17 | 30.37 |  |  |  |
|  |  |  | 1 | 27.26 | 3.18 | 30.44 |  |  |  |
|  |  |  | 2 | 13.83 | 1.86 | 14.69 | 1.97108 | 1.70968 | 2.07216 |
|  |  | 3 | 1 | 27.25 | 3.18 | 30.43 |  | 1.70968 | 2.07216 |
|  |  |  | 2 | 13.84 | 2.25 | 16.09 | 1.96893 | 1.41333 | 1.89124 |
|  |  |  | 3 | 2.28 | 2.01 | 11.29 | 2.93642 | 1.58209 | 2.69531 |
|  |  | 4 | 1 | 27.50 | 3.19 | 30.69 |  |  |  |
|  |  |  | 2 | 13.76 | 1.82 | 15.58 | 1.99855 | 1.75275 | 1.96983 |
|  |  |  | 3 | 9.33 | 2.30 | 11.63 | 2.94748 | 1.38696 | 2.63887 |
|  |  |  | 4 | 6.95 | 1.76 | 8.71 | 3.95683 | 1.81250 | 3.52354 |

TABLE $6.16(\mathrm{C})$
The timing results obtained from the implementation of the Version $1 I$ parallel digit sort using data size equal to 2048 .

| Program | No. of <br> processors | Total Time (sec.) | Speed-up | Parallel Paths in |  | SDO | PCO | Critical Section |  | Total <br> Critical <br> Section Time <br> (sec.) |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  |  |  | Sort Part | Distribution Part |  |  | Waiting Cycles | No. of Accesses |  |
| $\begin{aligned} & \text { Version I } \\ & \text { (size 1024) } \end{aligned}$ | 1 | 8.77 | 1.0 | 67 | 2 | $1.14 \%$ | 1.28\% | - | - |  |
|  | 2 | 4.95 | 1.77172 | 32,36 | 2,1 |  |  | - | - |  |
|  | 3 | 3.65 | 2.40274 | 25,19, | 2,1,1 |  |  | - | - |  |
|  |  |  |  | 25 |  |  |  |  |  |  |
|  | 4 | 3.00 | 2.92333 | $\begin{aligned} & 19,16, \\ & 16,19 \end{aligned}$ | 2,1,1,1 |  |  | - | - |  |
| $\begin{aligned} & \text { Version I } \\ & \text { (size 2048) } \end{aligned}$ | 1 | 29.32 | 1.0 | 67 | 2 | $1.10 \%$ | 0.99\% | - | - |  |
|  | 2 | 15.75 | 1.86159 | 36,32 | 2,1 |  |  | - | - |  |
|  | 3 | 11.23 | 2.61086 | 25,23, | 2,1,1 |  |  | - | - |  |
|  |  |  |  | 21 |  |  |  |  |  |  |
|  | 4 | 8.93 | 3.28700 | 19,16, | 2,1,1,1 |  |  | - | - |  |
|  |  |  |  | 18,17 |  |  |  |  |  |  |
| $\begin{aligned} & \text { Version II } \\ & \text { (size 1024) } \\ & \text { (4 paths) } \end{aligned}$ | 1 | 9.58 | 1.0 | 67 | 7 | 1.02\% | 1.06\% | 0 | 1024 | 0.82 |
|  | 2 | 4.87 | 1.96715 | 32,36 | 5,3 |  |  | 30,31 | 512,512 | 0.85 |
|  | 3 | 3.83 | 2.50131 | 25,19, | 5,2,2 |  |  | 307,303, | 512,256, | 0.74 |
|  |  |  |  | 25 |  |  |  | 297 | 256 |  |
|  | 4 | 2.90 | 3.30345 | 19,17, | 4,2,2,2 |  |  | 454,458, | 256,256, | 0.70 |
|  |  |  |  | 17,17 |  |  |  | 465,458 | 256,256 |  |
| Version II (size 2048) (4 paths) | 1 | 30.69 | 1.0 | 67 | 7 | 0.93\% | 0.82\% | 0 | 2048 | 1.64 |
|  | 2 | 15.58 | 1.96983 | 36,32 | 5,3 |  |  | 359,379 | 1024,1024 | 1.23 |
|  | 3 | 11.63 | 2.63887 | 25,23, | 5,2,2 |  |  | 711,708, | 1024,512, | 1.59 |
|  |  |  |  | 21 |  |  |  | 699 | 512 |  |
|  | 4 | 8.71 | 3.52354 | 18,16, | 4,2,2,2 |  |  | 924,919, | 512,512, | 1.41 |
|  |  |  |  | 19,17 |  |  |  | 919,918 | 512,512 |  |

TABLE 6.17: Performance analysis of the parallel digit sort algorithms

### 6.3 THE SEARCHING ALGORITHMS

Searching might be called the storage and retrieval of information, or it might simply be called table-lookup. By searching, one usually means the operation of locating a specific item in a given sequence of N items, i.e. to find the data that has been stored with a given identification. In general, we shall suppose that a set of N records has been stored, and the problem is to locate the appropriate one. Algorithms for searching are presented with a so-called argument $K$, and the problem is to locate which record has $K$ as its key. After the search is complete, two possibilities can arise:

Either the search was successful, having located the unique record containing $K$, or it was unsuccessful, having determined that $K$ is nowhere to be found.

Searching is the most time-consuming part of many programs, and the substitution of a good search method for a bad one often leads to a substantial increase in speed. It is often possible to arrange the data or the data structure so that the searching process can be eliminated entirely.

For one-dimensional search problems a lower bound of $\operatorname{logN}$ for searching a record amongst a set of $N$ records has been established.

In this section, two sequential search algorithms are implemented in parallel, these algorithms are the basic sequential search and the well known binary search. The experimental results of these two parallel algorithms are presented and analysed beside its performance analysis.

Given a table of records $R_{1}, R_{2}, \ldots, R_{N}$ whose respective keys are $K_{1}, K_{2}, \ldots, K_{N}$. Given an argument $K$, the search consists of a comparison between the argument $K$ and the key field ( $K_{i}$ ) associated with each
record $R_{1}, R_{2}, \ldots, R_{N}$, and action is taken based upon the result of comparisons; the search succeeds when $K_{i}=K$, for $i=1,2, \ldots, N$.

Without prior knowledge about the records, they must be assumed to be unordered, uniformly probable, and uniformly accessible. In this simple case, each access permits the examination of a single record, so that $\frac{1}{N}$ of the possibilities can be eliminated with each access. If there are $P$ processors, then $P$ accesses per cycle are permitted in parallel. In this case $\frac{P}{N}$ of the candidate items can be examined each cycle, including the single processor as a special case ( $p=1$ ).

### 6.3.1 Sequential Search (Unordered Table) (Knuth, 1973)

If the data is not ordered, then there are no preferred places to look for them. This means that in order to locate the target item $K$ in a field of similar items an exhaustive procedure must be used.

Given a table of records $R_{1}, R_{2}, \ldots, R_{N}$ whose respective keys are $K_{1}, K_{2}, \ldots, K_{N}$ this algorithm searches for a given argument $K$. We assume that $N \geqslant 1$.
(1) (Initialize) Set $i=1$, and set $K_{N+1}=K$,
(2) (Compare) If $K=K_{i}$, go to step (4),
(3) (Advance) Increase i by 1 , and return to step (2),
(4) (End of file) If $i \leqslant N$, the algorithm terminates successfully, otherwise it terminates unsuccessfully (i=N+1).

In this algorithm a dumy record $R_{N+1}$ is used at the end of the file. If every input key occurs with equal probability, the average value of the number of key comparisons (C) in a successful search will be:

$$
\begin{align*}
C & =\frac{1+2+\ldots+N}{N} \\
& =\frac{N(N+1) / 2}{N} \\
& =\frac{1}{2}(N+1) . \tag{6.3.1}
\end{align*}
$$

While for an unsuccessful search the number of comparisons will be equal to,

$$
\begin{equation*}
\mathrm{C}=\mathrm{N} . \tag{6.3.2}
\end{equation*}
$$

## Parallel Sequential Search

The sequential search algorithm has been implemented in parallel where $P$ parallel processors co-operate to search the whole list of numbers to locate a record with the required key. The implementation of the sequential search on the NEPTUNE system was carried out as follows:-

The input data set of N elements is partitioned into M subsets of size $\left(\frac{N}{M}\right)$ each, so that the first $\left(\frac{N}{M}\right)$ elements are allocated into the first subset and the next $\left(\frac{N}{M}\right)$ elements into the second subset and so on. We assume that $M \geqslant P$, where $P$ is the number of the available processors. Each subset is searched independently of the other subsets. The search is carried out in exactly the same manner as in the sequential form which was described in the algorithm. A flag is set up when the searched key is found in any of the subsets to prevent the remaining subsets to continue their search. 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 searching procedure is complete when the target item is found (i.e., the search successful) or when all the subsets have been searched (i.e., the search unsuccessful).

The parallel implementation of the sequential search algorithm on the NEPTUNE system is programmed in Program (6.7) and the experimental results obtained from that implementation are shown in Tables $6.18 a, b, c$ and $d$. These results are obtained for the input data size (N) equal to 9216 using different subsets (M) where the search is carried out for different keys. The searched keys are located within the input set of elements in different positions where the keys in Tables 6.18 a,b,c are in locations $2500,5000,7500$ respectively. While the results in Table 6.18 d are obtained when the search key is not in the input file, i.e. all the elements within the input file need to be searched.

From these results we notice that when one processor has been used the search time is increased as the number of subgroups (M) is increased. This is due to the overheads incurred from the creation of more parallel paths. While when using more than one processor the optimum (lowest) search time needed depends on the number of subgroups (M) used. For example, to search for the key 0.97599792 , the lowest time obtained when the input data set ( $N$ ) is partitioned into 16,32 and 64 subsets using 2,3 and 4 processors, respectively. This is due to the way in which the parallel implementation is carried out and the usage of the "FLAG" to prevent other processors from continuing their search when the search key is found in any subset, so the search time is dependent on the location of the key within each subset. For the speed-up results, we can generally say that the best results are obtained when the number of subsets (M) is the one that gives the optimal search time and this is because at that value of $M$ the cooperative processors are fully utilized.

| No. of Subset (M) | Processors |  |  |  |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 1 |  | 2 |  |  | 3 |  |  | 4 |  |  |
|  | $\begin{aligned} & \text { Time } \\ & (\mathrm{sec}) \end{aligned}$ | Total Paths | Time <br> (sec) | Total <br> Paths | Speed-up | Time <br> (sec) | Total <br> Paths | Speed-up | $\begin{aligned} & \text { Time } \\ & \text { (sec) } \end{aligned}$ | Total <br> Paths | Speed-up |
| 4 | 1.51 | 7 | 1.43 | 4,4 | 1.06 | 1.45 | 4,2,3 | 1.04 | 1.45 | 4,2,2,2 | 1.04 |
| 8 | 1.52 | 11 | 1.46 | 9,3 | 1.04 | 0.74 | 4,7,2 | 2.05 | 0.75 | 4,2,2,6 | 2.03 |
| 16 | 1.53 | 19 | 1.1 | 16,4 | 1.39 | 0.75 | 5,13,3 | 2.04 | 0.75 | 13,3,3,3 | 2.04 |
| 32 | 1.54 | 35 | 0.92 | 30,6 | 1.67 | 0.56 | 6,4,27 | 2,75 | 0.56 | 26,4,4 A | 2.75 |
| 64 | 1.57 | 67 | 0.82 | 12,56 | 1.92 | 0.56 | 9,7,53 | 2.80 | 0.47 | $\begin{gathered} 8,6,29 \\ 27 \end{gathered}$ | 3.34 |
| 128 | 1.64 | 131 | 0.88 | 69,63 | 1.86 | 0.62 | $\begin{gathered} 32,51 \\ 50 \end{gathered}$ | 2.65 | 0.49 | $\begin{array}{r} 24,45 \\ 21,44 \end{array}$ | 3.35 |
| 256 | 1.75 | 259 | 0.99 | $\begin{array}{r} 133, \\ 127 \end{array}$ | 1.77 | 0.72 | $\begin{gathered} 84,80 \\ 97 \end{gathered}$ | 2.43 | 0.59 | $\begin{array}{r} 61,66, \\ 64,71 \end{array}$ | 2.97 |
| 512 | 1.99 | 515 | 1.21 | $\begin{array}{r} 265, \\ 251 \end{array}$ | 1.65 | 0.93 | $\begin{gathered} 170,174 \\ 173 \end{gathered}$ | 2.14 | 0.76 | $\begin{aligned} & 129,131, \\ & 129,129 \end{aligned}$ | 2.62 |

(a) Results obtained when the search key $=0.61517334$

| No. of Subset (M) | Processors |  |  |  |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 1 |  | 2 |  |  | 3 |  |  | 4 |  |  |
|  | $\begin{aligned} & \text { Time } \\ & \text { (sec) } \end{aligned}$ | Total <br> Paths | Time <br> (sec) | Total <br> Paths | Speed-up | Time <br> (sec) | Total <br> Paths | Speed-up | Time (sec) | Total <br> Paths | Speed-up |
| 4 | 3.03 | 7 | 2.86 | 5,3 | 1.06 | 1.44 | 4,3,2 | 2.1 | 1.45 | 4,2,2,2 | 2.09 |
| 8 | 3.04 | 11 | 1.64 | 6,6 | 1.85 | 1.42 | 5,3,5 | 2.14 | 1.45 | 5,3,3,3 | 2.1 |
| 16 | 3.04 | 19 | 1.81 | 14,6 | 1.68 | 1.09 | 6,11A | 2.79 | 1.09 | 6,8,4,4 | 2.79 |
| 32 | 3.05 | 35 | 1.61 | 12,24 | 1.89 | 1.1 | 9,21,7 | 2.77 | 0.92 | 20,6,6,6 | 3.32 |
| 64 | 3.09 | 67 | 1.64 | 49,19 | 1.88 | 1.1 | $\begin{gathered} 15,13 \\ 41 \end{gathered}$ | 2.81 | 0.84 | $\begin{gathered} 12,10,38 \\ 10 \end{gathered}$ | 3.68 |
| 128 | 3.15 | 131 | 1.65 | 80,52 | 1.91 | 1.12 | $\begin{array}{r} 33,48 \\ 52 \end{array}$ | 2.81 | 0.88 | $32,28,27$ | 3.58 |
| 256 | 3.28 | 259 | 1.72 | $\begin{array}{r} 121 \\ 139 \end{array}$ | 1.91 | 1.19 | $\begin{gathered} 90,88 \\ 83 \end{gathered}$ | 2.76 | 0.94 | $\begin{gathered} 68,71,62 \\ 61 \end{gathered}$ | 3.49 |
| 512 | 3.54 | 515 | 1.90 | $\begin{array}{r} 254, \\ 262 \end{array}$ | 1.86 | 1.35 | $\begin{gathered} 172,178 \\ 167 \end{gathered}$ | 2.62 | 1.08 | $\begin{aligned} & 131,128 \\ & 132,127 \end{aligned}$ | 3.28 |

(b) Results obtained when the search key $=0.97599792$

| No. of subset <br> (M) | Processors |  |  |  |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 1 |  | 2 |  |  | 3 |  |  | 4 |  |  |
|  | Time <br> (sec) | Total <br> Paths | Time <br> (sec) | Total <br> Paths | Speed-up | Time <br> (sec) | Total Paths | Speed-up | Time (sec) | Total <br> Paths | Speed-up |
| 4 | 4.57 | 7 | 2.85 | 5,3 | 1.6 | 1.76 | 5,2,2 | 2.6 | 1.45 | 4,2,2,2 | 3.15 |
| 8 | 4.58 | 11 | 2.89 | 7,5 | 1.59 | 2.16 | 6,4,3 | 2.12 | 1.45 | 5,3,3,3 | 3.16 |
| 16 | 4.59 | 19 | 2.54 | 12,8 | 1.81 | 1.81 | 9,6,6 | 2.54 | 1.46 | 7,5,5,5 | 3.14 |
| 32 | 4.6 | 35 | 2.39 | 18,8 | 1.93 | 1.64 | $\begin{gathered} 12,10 \\ 15 \end{gathered}$ | 2.81 | 1.28 | $\begin{gathered} 10,8,8 \\ 12 \end{gathered}$ | 3.59 |
| 64 | 4.64 | 67 | 2.41 | 35,33 | 1.93 | 1.65 | $\begin{gathered} 31,19 \\ 19 \end{gathered}$ | 2.81 | 1.19 | $\begin{aligned} & 17,14, \\ & 24,15 \end{aligned}$ | 3.9 |
| 128 | 4.71 | 131 | 2.39 | 67,65 | 1.97 | 1.63 | $\begin{gathered} 46,36, \\ 51 \end{gathered}$ | 2.89 | 1.22 | $\begin{aligned} & 34,41, \\ & 28,31 \end{aligned}$ | 3.86 |
| 256 | 4.84 | 259 | 2.48 | 140,120 | 1.95 | 1.68 | $\begin{gathered} 90,93, \\ 78 \end{gathered}$ | 2.88 | 1.28 | $\begin{aligned} & 64,71, \\ & 64,63 \end{aligned}$ | 3.78 |
| 512 | 5.11 | 515 | 2.63 | 256,260 | 1.94 | 1.79 | $\begin{aligned} & 180, \\ & 168,169 \end{aligned}$ | 2.86 | 1.38 | $\begin{aligned} & 131,130 \\ & 126,131 \end{aligned}$ | 3.70 |

(c) Results obtained when the search key $=0.52330017$

| No. of Subset (M) | Processors |  |  |  |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 1 |  | 2 |  |  | 3 |  |  | 4 |  |  |
|  | Time <br> (sec) | Total Paths | Time <br> (sec) | Total <br> Paths | Speed-up | Time <br> (sec) | Total <br> Paths | Speed-up | Time <br> (sec) | Total <br> Paths | Speed-up |
| 4 | 5.63 | 7 | 2.87 | 5,3 | 1.96 | 2.83 | 5,2,2 | 1.99 | 1.45 | 4,2,2,2 | 3.88 |
| 8 | 5.64 | 11 | 2.87 | 7,5 | 1.97 | 2.15 | 6,3,4 | 2.62 | 1.46 | 5,3,3,3 | 3.86 |
| 16 | 5.65 | 19 | 2.87 | 11,9 | 1.97 | 2.13 | 9,6,6 | 2.65 | 1.45 | 7,5,5,5 | 3.89 |
| 32 | 5.66 | 35 | 2.88 | 19,17 | 1.97 | 1.98 | $\begin{gathered} 14,11 \\ 12 \end{gathered}$ | 2.86 | 1.45 | $11,9,9$, 9 | 3.9 |
| 64 | 5.7 | 67 | 2.90 | 35,33 | 1.97 | 1.97 | $\begin{gathered} 25,22 \\ 22 \end{gathered}$ | 2.89 | 1.46 | $\begin{aligned} & 19,17 \\ & 17,17 \end{aligned}$ | 3.9 |
| 128 | 5.77 | 131 | 2.93 | 67,65 | 1.97 | 1.97 | $\begin{gathered} 46,44, \\ 43 \end{gathered}$ | 2.93 | 1.48 | $\begin{aligned} & 35,33, \\ & 33,33 \end{aligned}$ | 3.9 |
| 256 | 5.91 | 259 | 2.99 | 131,129 | 1.98 | 1.99 | $\begin{gathered} 89,86 \\ 86 \end{gathered}$ | 2.97 | 1.51 | $\begin{aligned} & 67,65 \\ & 65,65 \end{aligned}$ | 3.91 |
| 512 | 6.91 | 515 | 3.12 | 259,257 | 1.98 | 2.08 | $\begin{gathered} 175,171 \\ 171 \end{gathered}$ | 2.98 | 1.58 | $\begin{aligned} & 132,128, \\ & 128,130 \end{aligned}$ | 3.92 |

(d) Results obtained when the search key $=0.99998877$

TABLE 6.18: The experimental timing results with the total number of parallel paths run by each processor obtained from searching an input data file of size 9216 elements.

From the results on Tables (6.18) the optimal search time for each key using different processors are tabulated in Table (6.19). From these results it is clear that there is a relation between the key position within the input list and the speed-up factors obtained. When the key is positioned at the end of the list, then clearly all $P$ processors are actively engaged in searching the list and the speed-up figures obtained reflect this. The deviation from $P$ measures the amount of overheads incurred in path conflicts (see Table 6.18d). When the key is positioned at the beginning of the list then clearly the speed tup figures obtained reflect the amount of searching to be done (i.e. there is not sufficient work involved for all 4 processors to be fully utilised or engaged) (see Table 6.18a). Intermediate key positions suggest that results in between these two extremes will be obtained and is entirely dependent on the position of the key in the list to be searched as to whether all the processors can be made active.

| Key | Processors |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 1 | 2 |  | 3 |  | 4 |  |
|  | Time | Time | Speed-up | Time | Speedup | Time | Speed up |
| 0.61517334 | 1.51 | 0.82 | 1.84 | 0.56 | 2.70 | 0.47 | 3.21 |
| 0.97599792 | 3.03 | 1.61 | 1.88 | 1.10 | 2.76 | 0.84 | 3.61 |
| 0.5233017 | 4.57 | 2.39 | 1.91 | 1.63 | 2.80 | 1.19 | 3.84 |
| 0.99998877 | 5.63 | 2.87 | 1.96 | 1.97 | 2.86 | 1.45 | 3.88 |

TABLE 6.19: The optimum timing results with its speed-up factors

Now we consider the total complexity of the sequential search algorithm when it is run on one processor (sequential machine) and when it is run on a $P$ processor system (parallel machine). Knuth [1973]
shows that if every input key occurs with equal probability, the average value of the number of key comparisons $C$ in a successful and an unsucessful search are shown in equations (6.3.1) and (6.3.2) respectively and these are,

$$
C= \begin{cases}\frac{1}{2}(N+1) & \text { if the search is successful } \\ N & \text { if the search is unsuccessful }\end{cases}
$$

However for a successful search, in our parallel implementation of the sequential search, the input to be searched is partitioned into $M$ subsets or paths with $N$ divisible by $M$ and $M \geqslant P$ (where $P=n u m b e r$ of available processors), then in each subset $N / M$ elements are stored. So, if the paths are carried out on one processor, then by applying equation (6.3.1) above in each subset there will be an average $C_{1}=\frac{1}{2}\left(\frac{N}{M}+1\right)$ comparisons. And for all the $M$ subsets the same idea is used, and hence, the total number of comparisons

$$
\begin{align*}
& =C_{1}+2 C_{1}+3 C_{1}+\ldots+M C_{1} \\
& =c_{1}(1+2+3+\ldots+M) \\
& =\frac{1}{2} M(M+1) \cdot C_{1} \tag{6.3.3}
\end{align*}
$$

so the average number of comparisons in all the $M$ subsets, will be

$$
\begin{align*}
C_{1 s} & =\frac{1}{2}(M+1) C_{1} \\
& =\frac{1}{2}(M+1)\left(\frac{1}{2}\left(\frac{N}{M}+1\right)\right) \\
& =\frac{1}{4 M}(M+1)(N+M) . \tag{6.3.4}
\end{align*}
$$

Meanwhile, when the algorithm is run in parallel with $P$ processors, $\left\lceil\frac{M}{P}\right\rceil$ paths have to be carried out by each processor. Thus,

$$
\begin{align*}
C_{p s} & =\left\lceil\frac{M}{P}\right\rceil\left[\frac{1}{4 M}(M+1)(N+M)\right] \\
& \leqslant \frac{1}{4} \frac{M N}{P}+\frac{1}{4} \frac{M^{2}}{P}+\frac{1}{4} \frac{N}{P}+\frac{1}{4} \frac{M}{P}+1 \tag{6,3.5}
\end{align*}
$$

Since we are interested in determining how much more efficient the
algorithm is when it is applied to a parallel computer, therefore we measure the speed-up ratio $S_{p s}(M)$ for the sequential search with $M$ subsets. Thus,

$$
\begin{align*}
S_{p s} & =\frac{C_{1 s}}{C_{p s}} \\
& \approx \frac{4\left(M N+M^{2}+N+M\right) P}{4 M\left(M N+M^{2}+N+M+4 P\right)} \\
& \approx P\left(\frac{M N+M^{2}+M^{2} N+M^{3}+4 P M-4 P M-M^{3}-M^{2} N+M+N}{M^{2} N+M^{3}+M N+M^{2}+4 P M}\right) \\
& \approx P\left[1-\left(\frac{4 P M+M^{3}+M^{2} N-M-N}{M^{2} N+M^{3}+M N+M^{2}+4 P M}\right)\right]  \tag{6.3.6}\\
S_{p s} & \approx O(P)
\end{align*}
$$

While in an unsuccessful search, the corresponding values are,

$$
\begin{equation*}
C_{1 s}=M\left(\frac{N}{M}\right)=N, \tag{6.3.7}
\end{equation*}
$$

and

$$
\begin{align*}
C_{p s} & =\left\lceil\frac{M}{P}\right\rceil\left(\frac{N}{M}\right) \\
& \leqslant \frac{N}{P}+1 . \tag{6.3.8}
\end{align*}
$$

Thus,

$$
\begin{align*}
s_{p s} & =\frac{C_{1 s}}{C_{p s}} \\
& \approx \frac{N}{\frac{N}{P}+1} \\
& \approx P\left(\frac{N}{N+P}\right) \\
& \approx P\left(1-\frac{P}{N+P}\right) \\
& \approx P\left(1-\frac{1}{\frac{N}{P}+1}\right)  \tag{6.3.9}\\
S_{p s} & \approx O(P) \quad .
\end{align*}
$$

It can be easily noticed that from equation (6.3.9) the linear speedup is easily achieved, especially for large $N$ and this is what we
obtain from the experimental results because the whole list is searched and the processors are fully utilized where a linear speed-up is achieved.

For a further performance analysis of the parallel sequential search and with reference to Chapter 4 , Table (6.20) shows the resource demands required when the mean rate of access to shared data and parallel paths are represented besides the parallel control overheads (PCO) and shared data overheads (SDO) are calculated. Both SDO and PCO are calculated when the algorithm was run on the NEPTUNE system when the search key was equal to 0.99998877 (Table 6.18d). From the results in Table ( 6.20 ) we notice that the parallel control overheads are increased as the number of parallel paths (subsets) are increased and this is what we get generally from both the experimental and the expected demands. Also these overheads show the reasons why the speedup obtained for the parallel implementation (Table 6.19) are not of $O(P)$.

To conclude this section, the sequential search algorithm is time consuming when run on a sequential machine (especially for large input size of data). However we find its parallel implementation on an MIMD type machine is good and this is clear from the results shown in Table (6.19) where all the processors are fully utilized and an acceptable speed-up factor is obtained.

The results in Table (6.19) are diagrammatically represented in Figure 6.6 where a linear speed-up is obtained.


The speed-up results obtained from the parallel sequential search algorithm using the data in Table 6.19

| Input Size <br> (N) | No. of Subsets (M) | Performance Measurement |  | Resource Demands |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  |  |  | Parallel Path |  | Shared Data |  |
|  |  | PCO | SDO | Access rate | Overheads | Access rate | Overheads |
| 9216 | 4 | 0.355\% | 0.355\% | $\begin{aligned} & 1:\left(13 * \frac{N}{M}\right) \\ & \text { flops } \end{aligned}$ | 0.006\% | $\begin{aligned} & (6: 10) \\ & \text { flops } \end{aligned}$ | 0.063\% |
|  | 8 | $0.401 \%$ | $0.355 \%$ |  | 0.0118 |  |  |
|  | 16 | $0.531 \%$ | $0.531 \%$ |  | 0.022\% |  |  |
|  | 32 | 0.707\% | 0.353\% |  | $0.045 \%$ |  |  |
|  | 64 | 1.2288 | $0.526 \%$ |  | 0.089\% |  |  |
|  | 128 | $2.08 \%$ | 0.520\% |  | 0.178\% |  |  |
|  | 256 | 4.061\% | $0.338 \%$ |  | 0.356\% |  |  |
|  | 512 | 7.4318 | 0.3238 |  | 0.712\% |  |  |

TABLE 6.20: Performance measurement and resource demands of the parallel sequential search algorithm.

### 6.3.2 Binary Search

If the input data is stored with some prescribed order then another method may be used to search for a specific key ( $K$ ) known as the binary search method. The previous sequential search is essentially limited to a two-way decision $\left(K_{i}=K_{i}\right.$ vs. $\left.K \neq K_{i}\right)$, while in the binary search method the search continues in three different ways, depending on whether $K<K_{i}, K=K_{i}$ or $K>K_{i}$. Hence we start comparing $K$ with the middle key in the table (the input data set); the result of this probe tells us which half of the table should be searched next, and the same procedure can be used again, comparing $K$ to the middle key of the selected half, etc. After (log N) comparisons, we will have found the key or we will have established that it is not present.

Sequential Binary Search Algorithm (Knuth, 1973)
Given a table of records $R_{1}, R_{2}, \ldots, R_{N}$ whose keys are in increasing order $K_{1}<\mathrm{K}_{2}<\ldots<\mathrm{K}_{\mathrm{N}}$, this algorithm searches for a given argument K . as follows:
(1) (Initialize) set $\ell=1, u=N$.
(2) (Obtain mid-point)

If $u<\ell$, the algorithm terminates unsuccessfully, otherwise, set
$i=\lfloor(\ell+u) \mid 2\rfloor$, the approximate midpoint of the relevant table area.
(3) (Compare) If $K<K_{i}$, goto step (4)

If $K>K_{i}$, goto step (5)
If $K=K_{i}$, the algorithm terminates successfully.
(4) (Adjust u) Set $u=i-1$, goto step (2).
(5) (Adjust $\ell)$ set $\ell=i+1$, goto step (2).

To make the binary search algorithm more clear, we can represent the binary search algorithm as a binary decision tree as shown in Figure 6.7. For the case $N=16$, the first comparison made by the algorithm is $K: K_{8}$; this is represented by the root node (8) in the Figure 6.7. Then, if $K<K_{8}$, the algorithm follows the left subtree, comparing $K$ to $K_{4}$, etc; Similarly if $K>K_{8}$, the right subtree is used. An unsuccessful search will lead to one of the "external" square node numbered 0 through $N$ for example, we reach node 6 if and only if $\mathrm{K}_{6}<\mathrm{K}<\mathrm{K}_{7}$.

Knuth [1973] and Baase [1978] shows that the number of comparisons (C) done by the binary search algorithm, in the worst case, for a list of $N$ entries, is,

$$
\begin{equation*}
C=\lfloor\log N\rfloor+1 \tag{6.3.10}
\end{equation*}
$$

for $N \geqslant 1$.


FIGURE 6.7: A binary decision tree for $\mathrm{N}=16$

## Parallel Binary Search

Consider the effects of using more than one independent processor (searcher) to occupy the actual or potential nodes of a binary search. In the binary search algorithm which half of the table to look at next is the output of one level of the algorithm's operation. If two processors are available, two cells can be examined at once; half the time the second processor will have provided valuable lookahead information, half the time its efforts will have been squandered in the wrong direction.

Another way to use the processors available might be to use them at each step to subdivide the input by more than $\frac{1}{2}$; i.e., to change
the binary search to a ternary one, i.e. 3 etc. This leads to the multisection algorithm.

A third way is to partition the input to be searched into a number of subgroups then apply the sequential binary search within each partition and a broadcast is sent if the target item is found in any one of the subgroups. The number of partitions may be greater than or equal to number of available processors.

The third method has been implemented practically, because we overcome the disadvantage of the first method by broadcasting the result if the target item has been found in any partition and no further search will be needed. In addition there is no need for the input items to be partitioned into exactly the same number as available processors as previously.

Now for the complexity of the parallel binary search, we suppose that if $N$ is the size of the input to be searched which is partitioned into $M$ (M divisible by $N$ ) subsets (or paths) with $M \geqslant P$ (where $P$ is equal to the number of processors available), then in each subset $N / M$ elements are stored. If all the paths are carried out on one processor, then by applying the binary sequential search formula (equation (6.3.10)), we get

$$
\begin{align*}
C_{1 s} & =M\left[\log \frac{N}{M}+1\right] \\
& \leqslant M\left(\log \frac{N}{M}+1\right) . \tag{6.3.11}
\end{align*}
$$

Meanwhile, when the algorithm is run in parallel with $P$ processors, $\left\lceil\frac{M}{\mathrm{P}}\right\rceil$ paths have to be carried out by each processor. Thus, we have the result,

$$
\begin{align*}
C_{p s} & =\left\lceil\frac{M}{P}\right\rceil\left\lfloor\log \frac{N}{M}+1\right\rfloor \\
& \leqslant \frac{M}{P}\left(\log \frac{N}{M}+1\right)+1 \tag{6.3.12}
\end{align*}
$$

We are interested in discovering how much more efficient the algorithm is when run on a parallel computer, therefore we measure the speed-up ratio $S_{p s}(M)$ for the sequential search with $M$ subsets. Thus

$$
\begin{align*}
s_{p s} & =\frac{C_{1 s}}{C_{p s}} \\
& \approx \frac{M\left(\log \frac{N}{M}+1\right)}{\frac{M}{P}\left(\log \frac{N}{M}+1\right)+1} \\
& \approx \frac{P M\left(\log \frac{N}{M}+1\right)}{M\left(\log \frac{N}{M}+1\right)+P} \\
& \approx P\left(1-\frac{P}{M\left(\log \frac{N}{M}+1\right)+P}\right) \tag{6.3.13}
\end{align*}
$$

which is of $O(P)$. This means that the optimum linear speed-up is achieved for the unsuccessful case also. It can be seen that greater efficiency can be obtained the larger $M$ is chosen.

The parallel implementation of the binary search algorithm on the NEPTUNE system has been programmed in Program (6.8) and the experimental results obtained from running this program are shown in Table (6.21). The results are obtained for the input size equal to 9216 using different numbers of subsets (M) to search for different keys. The results in Table (6.21) are taken as an average of many runs. For the input data size of 9216 , the actual search time obtained from the parallel search algorithm is very small and because this is the highest input data size which can be used (due to the restriction of the NEPTUNE system) the appropriate parts in Program 6.8 have been repeated 50 times to make the results measurable.

From the results in Table (6.21) it can be noticed that the search time is increased as the number of subsets (parallel paths) is
increased and this is due to the overheads incurred by the system such as the generation of the parallel paths and the communication between the processors. The speed-up factors are also increased as the number of subsets is increased and this is due to the fact that the co-operative processors are more utilized with the high number of subsets available. Also from the experimental results we notice that, unlike the parallel sequential search algorithm there is no relation between the position of the searched key and the search time, i.e. it is not necessary for the key located at the end of the input data set (subset) requires the highest search time. This is because in this method we always start searching at the middle of each subset, and the key with the highest number of comparisons requires the highest search time. We notice also the speed-up ratios obtained from the implementation of the parallel binary search algorithm are generally small for the case when the number of subsets are less than or equal to 32 ( $M \leqslant 32$ ). This is because the percentage of the overheads (parallel control, processor communication and shared data access) to the search time are high in those cases (i.e. M<32).

Now, the performance analysis of this method is predicted together with their performance measurements when run on the NEPTUNE system. With reference to Chapter 4, we measure the losses due to the shared data and the parallel paths. For the shared data loss we have to consider how many accesses to the shared data per total number of operations carried by one path. While for the losses due to the parallel path control, we have to know the number of accesses made by the program to a path per total number of operations performed in the path. Table (6.22) shows the predicted resource demands and also
represents the performance measurements of the parallel binary search when run on the NEPTUNE system for the input size 9216 to search for the key 0.99998877 using different subsets (see Table 6.21). From the results in Table (6.22) we notice that in the resource demands the parallel overheads are increased as the number of parallel paths is increased. The experimental results confirm this prediction where the obtainable parallel control overheads (PCO) are increased as the number of parallel paths is increased and this is due to the extra overheads incurred by the system. We also notice that the percentage of the overheads are generally high and this is because the binary search time is low for our input data size.

To conclude this section we can say that the expected performance of the parallel binary search algorithm is good for very large input and this is clear from the results in Table (6.21) where the speed-up is very good (high) when the co-operating processors are fully utilized and this is what we expect from large input sizes.

| N | Key | No.of <br> Subsets <br> (M) | No. of Processors |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  |  | 1 | 2 |  | 3 |  | 4 |  |
|  |  |  | $\begin{gathered} \text { Time } \\ (\mathrm{sec}) \end{gathered}$ | Time <br> (sec) | Speed- up | $\begin{aligned} & \text { Time } \\ & \text { (sec) } \end{aligned}$ | Speedup | Time (sec) | Speedup |
| 9216 | 0.52330017 | 4 | 2.44 | 1.96 | 1.25 | 1.51 | 1.62 | 1.50 | 1.63 |
|  |  | 8 | 3.97 | 2.72 | 1.46 | 2.08 | 1.91 | 1.98 | 2.01 |
|  |  | 16 | 6.86 | 4.21 | 1.63 | 2.94 | 2.33 | 2.79 | 2.46 |
|  |  | 32 | 12.04 | 6.87 | 1.75 | 5.10 | 2.36 | 4.38 | 2.75 |
|  |  | 64 | 21.88 | 12.2 | 1.79 | 8.90 | 2.46 | 7.28 | 3.01 |
|  |  | 128 | 39.20 | 21.61 | 1.81 | 15.61 | 2.51 | 12.61 | 3.11 |
|  | 0.61517334 | 4 | 2.65 | 1.98 | 1.34 | 1.43 | 1.22 | 1.47 | 1.18 |
|  |  | 8 | 4.17 | 2.93 | 1.42 | 2.31 | 1.81 | 2.04 | 2.04 |
|  |  | 16 | 7.77 | 4.56 | 1.70 | 3.69 | 2.11 | 2.99 | 2.60 |
|  |  | 32 | 14.11 | 7.89 | 1.79 | 5.81 | 2.43 | 4.81 | 2.93 |
|  |  | 64 | 25.36 | 13.96 | 1.82 | 9.96 | 2.55 | 8.01 | 3.17 |
|  |  | 128 | 44.67 | 24.15 | 1.85 | 17.13 | 2.61 | 13.65 | 3.27 |
|  | 0.97599792 | 4 | 3.32 | 2.11 | 1.57 | 1.63 | 2.04 | 1.50 | 2.21 |
|  |  | 8 | 6.26 | 3.62 | 1.73 | 2.77 | 2.26 | 2.25 | 2.78 |
|  |  | 16 | 11.71 | 6.26 | 1.87 | 4.41 | 2.66 | 3.65 | 3.21 |
|  |  | 32 | 21.36 | 11.16 | 1.91 | 7.55 | 2.83 | 6.01 | 3.55 |
|  |  | 64 | 37.91 | 19.67 | 1.93 | 13.25 | 2.86 | 10.23 | 3.71 |
|  |  | 128 | 66.86 | 34.15 | 1.96 | 23.05 | 2.90 | 17.60 | 3.80 |
|  | 0.99998877 | 4 | 3.64 | 2.43 | 1.50 | 1.95 | 1.87 | 1.74 | 2.09 |
|  |  | 8 | 6.59 | 3.94 | 1.67 | 3.03 | 2.18 | 2.45 | 2.69 |
|  |  | 16 | 12.03 | 6.52 | 1.85 | 4.84 | 2.49 | 3.72 | 3.23 |
|  |  | 32 | 21.69 | 11.49 | 1.89 | 8.00 | 2.71 | 6.09 | 3.56 |
|  |  | 64 | 38.8 | 20.04 | 1.94 | 13.57 | 2.86 | 10.50 | 3.70 |
|  |  | 128 | 68.76 | 34.97 | 1.97 | 23.60 | 2.91 | 17.95 | 3.83 |

TABLE 6.21: The experimental results obtained from the parallel
implementation of the binary search algorithm

| Input Size <br> (N) | No. of Subsets (M) | Performance Measurement |  | Resource Demands |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  |  |  | Parallel Path |  | Shared Data |  |
|  |  | PCO | SDO | Access rate | Overheads | Access rate | Overheads |
| 9216 | 4 | 6.3198 | 0.550\% | (1:14 $\times$ | $3.541 \%$ | (4:13) flops | 0.032\% |
|  | 8 | 6.373\% | 0.455\% | $\left.\log \frac{N}{M}\right)$ flops | 3.889\% |  |  |
|  | 16 | 6.567\% | $0.333 \%$ |  | 4.312\% |  |  |
|  | 32 | 6.777\% | $0.323 \%$ |  | 4.841\% |  |  |
|  | 64 | 7.242\% | $0.361 \%$ |  | 5.515\% |  |  |
|  | 128 | 8.115\% | 0.422\% |  | 6.410\% |  |  |

TABLE 6.22: Performance measurement and resource demands of the parallel binary search algorithm

### 6.4 CONCLUSIONS

In this chapter, two parallel sorting and searching methods have been implemented on the NEPTUNE system. For the sorting methods, the first algorithm (the shell sort method) belongs to the comparative sorting class, while the second algorithm (the digit sort method) belongs to the distributive sorting class.

For the parallel shell algorithm, two versions have been programmed on the NEPTUNE system, where in version I the sort procedure was only used, while in Version II the merge procedure is used after the first pass of the sort procedure. In Version $I$, the parallel implementation is carried out by two approaches, where the difference between the two approaches is the way in which the distance of comparison is chosen. From the experimental results it was shown that the second approach needs less time for sorting and gives better speedup ratios than that of the first approach. This is due to the way in which the distance of comparison is chosen where fewer passes are needed in the second approach than that of the first one.

In Version II of parallel shell sort, the algorithm is carried out in two stages, these are the sorting and the merging stages. In the merge stage, two merge algorithms have been used, these are the 2 way and the odd-even merge algorithms. It is noticed that using the 2-way merge procedure gives less sorting time than that used in the odd-even merge procedure. This is due to the fact that the algorithm that used the 2 -way merge needs $\log$ m steps to sort the input data set (where $M$ is the number of subsets), while the algorithm that used the odd-even merge needs $M$ steps where $M>\log M$. Meanwhile, the speed-up
factors for the algorithm that used the odd-even merge procedure is better than that used in the 2-way merge. This is because the processors in the odd-even merge are more utilized, while in the $2-$ way merge the number of active processors are halved in each step of the algorithm, where only one processor is used in the final step while the remaining processors are idle. Generally we notice that the speed-up factors obtained from the parallel shell sort is not high (see Table 6.8) and this is due to the manner in which the shell sort algorithm performs its procedure, where from pass to pass the number of active processors are decreased, in other words the co-operative processors are not fully utilized.

For the digit sort algorithm, two parallel versions have been implemented on the NEPTUNE system. The two versions are the same except that the first part of the algorithm was implemented sequentially in the first version, while in the second version it was implemented in parallel. The experimental results show that for the first version the shortest sorting time is obtained when using the largest number of subsets, i.e., when the number of subsets is equal to 64 in our implementation (see Table 6.13). For the sort part only, the best efficiency (speed-up) is obtained when the number of subsets used is the largest. This is because the processors are fully utilized. We notice also that the total efficiency (speed-up) was not as good as the speed-up of the sort part only and this is because the distribution part is done sequentially which decreases the efficiency (as shown in Table 6.13). We also observe from the experimental results that better efficiency is obtained with the largest input data size and this is in agreement with the theoretical results (Maclaren [1966]).

The results obtained from Version II of the parallel digit sort algorithm also show that the best (shortest) sorting time is obtained when the number of subsets are the largest. The expected gain in using parallel distribution instead of the sequential one (as in Version I) is not good and this is due to the usage of the critical section while maintaining the links between the lists and the sublists. This means only one processor can be active within that section of the program while the rest of the processors are idle which greatly affects the algorithm's performance. In other words, the overheads for the critical section accesses are much more than the parallel gain in the algorithm (see Table 6.17). For both version I and II of the parallel digit sort methods we can generally say that Version II is more suitable for the MIMD type computers. This is clear from the utilization of the processors shown in Table 6.17.

For the searching algorithms, two parallel methods were implemented on the NEPTUNE system. These are the parallel sequential search and the parallel binary search methods.

For the parallel sequential search algorithm it is clear from the experimental results in Table 6.19 that the optimum results depend on the location of the search key within the input data set. The best efficiency (speed-up) is obtained when the key is located at the end of the file, i.e. all the processors are fully utilized. Also, from the experimental results we conclude that the parallel search algorithm is generally efficient for the MIMD type machine (see Table 6.19).

While for the parallel binary search algorithm, we notice that the search time is increased as the number of subsets (parallel paths) is increased and this is due to the overheads incurred by the system.

The efficiency of the parallel binary search algorithm is also increased as the number of subsets is increased where the processors are more efficiently utilized with the high number of subsets. In the parallel binary search algorithm we notice that the percentage of the overheads are generally high and this is because the parallel binary search time is low for our input data size (see Table 6.22). We expect that a better performance will be obtained when the input data size is very large and this is due to the fact that the processors with the large input sizes are more utilized (see Table 6.21).

## Chapter Seven

## SUMMARY AND CONCLUSIONS

The central theme of this thesis is to cover the design and analysis of asynchronous parallel algorithms that can be run on MIMD type computers and in particular the NEPTUNE system at Loughborough University.

In the first three introductory chapters, the fundamentals of parallel computer architectures, parallel programming principles and design of parallel algorithms have been introduced.

As computed processing power increases, parallel processing is considered as a natural and feasible approach to achieve these demands. Parallel computers have been classified in this thesis into various different types, each of which has its own characteristics and the types of problems and applications for which it is more suitable to solve. For the implementation of certain parallel algorithms, some difficulties may arise and should be taken into consideration such as the communication and the synchronisation between all the co-operating processors. These problems can be overcome either implicitly or explicitly as seen in Chapter 2.

In general, programming parallel systems is not as easy as that of uniprocessor systems and this had led to the parallelism being concealed on most existing MIMD computers. Since the MIMD-type computers consist of $P$ complete computers where $P$ independent computations can be supported simultaneously. Hence, the main problem in programming MIMD-type computers lies in making the computers co-operate efficiently. So that one problem can be appropriately partitioned amongst them to solve a given problem with greater speed than it could be solved on a uniprocessor. In order to make the MIMD computers competent, it is essential that the obtainable speed is of $O(P)$, in comparison with the smallest
possible sequential time achieved by the best method. This can be obtained by minimizing both the synchronisation and shared data overheads, which are directly dependent upon the overall computational scheduling.

The performance analysis of an algorithm is important from different points of view. Essentially, it can help to understand better the algorithm and sometimes to reveal further necessary improvements. In other words, the careful search required for a proper performance analysis often leads to a more efficient and a more correct implementation of algorithms. However, the more complicated the algorithm, the more difficult its performance analysis becomes.

The principle behind the performance analysis is that parallel processing 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. These demands are determined by the programs, while the availability and allocation algorithms are properties of the system.

In recent research promising results have been achieved by getting a better speed-up by the explicit use of parallelism through the program. Two types of algorithms have been implemented on the NEPTUNE system and have been studied in this thesis, these are the numerical and nonnumerical algorithm.

In Chapter 4, the parallel 9-point explicit block iterative method was developed and implemented on the NEPTUNE system. The implementation of the parallel 9-point and parallel 4-point block iterative methods were programmed using different versions and strategies involving
synchroneity and asynchroneity together with natural or red-black orderings. It is clear that the implementation of different strategies present different timings and losses when they are run on the NEPTUNE system. For both the parallel 9-point and parallel 4-point block iterative method, the implementation Version 2 gives better timing results in all the strategies considered and this is due to the way in which each block within each subset is evaluated. In Version 2 the number of operations required is less than that of Version 1 . Also in Version 2, a greater rate of convergence is achieved since the most recent values of some points are used in evaluating the remaining other points within each block. Also, from the different implemented strategies, the asynchronous implementation gives better results than the synchronous one. This is due to the overheads required at the end of each iteration in the synchronous implementation. Also, in the asynchronous implementation better results are obtained because the processors are almost always fully occupied and busy doing work most of the time.

The overheads of shared data and parallel path access were measured for all the strategies and the 9-point asynchronous version required less overheads which resulted in the best results being obtained.

It can be seen from the experimental results that the parallel 9point block iterative method takes less time than its corresponding parallel 4-point block iterative method when $\omega=1.0$, while when $\omega=\omega_{\text {opt' }}$, in general the two parallel methods take the same time. Therefore, the parallel 9-point block iterative method was chosen as best amonst the two parallel block methods. Also it can be seen from the experimental results, that the parallel 9-point block and 4-point block methods are
best suited for parallel implementation on a MIMD computer and this is due to the almost linear speed-up obtained from their implementation.

In Chapter 5, the parallel A.G.E. method has been developed where two strategies have been implemented and used to solve a linear and a non-linear boundary value problem. The two strategies were programmed on the NEPTUNE system using both the synchronous and asynchronous approach.

For the linear problem (Problem I), the best results were obtained when the problem is solved using Strategy I of the parallel A.G.E. method with the asynchronous approach. This is due to the total number of computational operations in Strategy II being higher than that of Strategy I and also there is the case that the old values are used while evaluating the next point using Strategy II.

For comparison reasons, the parallel versions of the Jacobi, Gauss-Seidel and S.O.R. iterative methods are implemented on the NEPTUNE system and used to solve the linear problem using the synchronous and asynchronous approaches. By comparing the results obtained from these implementations and those obtained from the parallel A.G.E. methods it is clear that the elapsed times using the parallel A.G.E. method gives better results in all the cases. This is because the number of iterations in the parallel Jacobi, Gauss-Seidel and S.O.R. methods are much higher than that of the parallel A.G.E. method, which means more total computational operations are required.

For the non-linear problem (Problem II), the results obtained from using the parallel A.G.E. method shows that the asynchronous approach gives better results than the synchronous approach and this is due to the synchronisation overheads occurring at the end of each iteration.

For comparison reasons, the parallel Jacobi, Gauss-Seidel and N.L.O.R. iterative methods are also used to solve Problem II. These results show that the parallel A.G.E. method gives better results in the case of the parallel Jacobi and Gauss-Seidel methods. This is because the number of iterations in the parallel Jacobi and GaussSeidel methods are higher than those of the parallel A.G.E. method, which means more computational operations are required to obtain a solution. While the parallel N.L.O.R. method gives the shortest timing results than those of the parallel A.G.E. method this is because more computational operations are required in the case of the parallel A.G.E. method than that of the parallel N.L.O.R. method.

From the experimental results of the parallel A.G.E. methods, the shared data and parallel control access are calculated and can be noticed in the case of the synchronous implementation to be higher than that of the asynchronous implementation. From the speed-up results obtained for both the linear and non-linear problem we notice that a greater speed-up is obtained in the non-linear problem and we conclude that the amount of computations carried out over the total overheads in the non-linear problem is greater than that of the linear problem.

From the experimental results obtained we notice that an almost linear speed-up is achievable and we can conclude that the parallel A.G.E. method is suitable for implementation on a MIMD computer.

In Chapter 6, two non-numerical algorithms have been implemented on the NEPTUNE system. These methods are concerned with parallel sorting and parallel searching.

For the parallel sort, two algorithms were implemented using different approaches: these are the parallel shell sort and the
parallel digit sort algorithms. In the parallel shell sort algorithms two versions were implemented. In both version I and II of the shell sort algorithm the created number of parallel paths are dependent on the way in which the distance of comparisons was chosen. Where the distance of comparison is large the number of parallel paths generated is large and vice versa.

In Version $I$ of the parallel shell sort algorithm only the shell sort procedure is used to get the final sorted list. While in version II, the parallel merge algorithm is used after the sort stage to obtain the final sorted output list. Two approaches have been used to implement Version I, where the difference between them is the way in which the distance of comparison was chosen. From the experimental results we noticed that the second approach gives better results (less time and better speed-up) than the first approach. This is because the number of passes in the second approach is less than that of the first approach and hence the total number of operations in the second approach is less.

In Version II, the parallel shell sort algorithm is carried out in two parts. In the first part the subgroups are sorted first, while in the second part the sorted subgroups are merged to obtain the final sorted list. Two different parallel merge algorithms were implemented in the merge part of Version II. These algorithms are the parallel 2way merge and the parallel odd-even reduction merge. From the experimental results we noticed that the sorting time using the 2 -way merge algorithm is less than the odd-even merge algorithm. This is because the 2-way merge needs only logM steps to merge the $M$ subgroups, while $M$ steps are required for the odd-even merge algorithm, where $M>\operatorname{logM}$.

From the speed-up results obtained from these two merge algorithms a higher speed-up was obtained in the case of the odd-even merge algorithm. This is because in the odd-even merge algorithm the processors are more utilized than in the 2 -way merge and this is due to the fact that in the 2-way merge algorithm the number of processors are halved at each step where only one processor is used in the final step and the rest of the processors are idle. Generally from the different implementations of the parallel shell sort algorithm, we noticed that the speed-up (efficiency) of the algorithms are not high, and this is because the way in which the shell sort algorithm carries out its procedure where the number of active processors are decreased and specially in the last pass where only one processor is used. In other words the co-operative processors are not fully utilized and from the performance analysis of the parallel shell sort algorithm the overheads of the parallel control and shared data access are relatively high which directly affects the performance of the algorithm.

For the digit sort algorithm, two parallel versions were implemented on the NEPTUNE system. In Version $I$, the first part of the algorithm (the presort or distribution part) was implemented sequentially, while in Version II the first part was implemented in parallel where the other parts of the algorithms are the same in both versions.

The experimental results obtained from the implementation of the parallel Version I shows that the sorting time decreases as the number of subgroups are increased, with the lowest time being obtained when the number of subgroups is equal to 64 in our input case. We also notice in Version I that the total (distribution and sort parts) speed-up is not as good as the sorting part only and this is because the distribution
part is done sequentially which affects the total speed-up. In Version I we observed from the experimental results that a better efficiency was obtained with the higher input size data.

For Version II of the parallel digit sort we also noticed that the lowest sorting time is obtained with the highest number of subsets. Because in Version II the first part (presort) of the algorithm was implemented in parallel, the expected gain is not good and this is due to the usage of the critical section while maintaining the links between the list and the sublists. This means the overheads of the critical section accesses are much more than the parallel gain in the algorithm.

Finally, for the parallel digit sort algorithm, we can say that Version II is more suitable for MIMD type computers and this is because the utilization of the processors in Version II is more than that in Version I.

For the search methods, two well known methods have been implemented in parallel, these methods are the parallel sequential search and the parallel binary search.

The experimental results obtained from the implementation of the parallel sequential sort shows that the optimum results is dependent on the location of the searched key within the input data set. The best efficiency (speed-up) was obtained when the key was located at the end of the file, i.e., all the processors are fully utilized. From the experimental results we can conclude that the parallel sequential search method is generally good for the MIMD type machine (see Table 6.19).

For the parallel binary search algorithm, we notice that the search time increases as the number of subsets (parallel paths) is increased and this is due to the overheads incurred by the system. The
efficiency of the algorithm is increased as the number of subsets is increased where the processors are more utilized with the higher number of subsets. In the parallel binary search algorithm, because the searching time is low for the input size used we noticed that the percentage of the overheads are generally high (see Table 6.22). From the experimental results we also expect that the performance will be improved for the very large input data gizes since the processors are more utilized for these cases.

## APPENDIX

Selected Computer Programs
$C$
$C$
$C$
$C$
$C$
$C$
$C$
$C$
$C$
$C$
$C$
$C$
$C$
$C$
$C$
$C$
$C$
$C$
THIS FROGFAM IMF'LEMENTS THE 9-FOINT ELOCK ITEFATIUE METHOLI WHEFE
THE SOR ITERATIUE METHOLI IS USEI. THIS IS AN ASYNCHRONOUS ALGORITHM
TO SOLUE THE 2-LIMENTIONAL IIFICHLET FROBLEM.THE LINES OF THE
MESH TO BE SOLVEI ARE FARTITIONED INTO NFATH SUBSETS SO THNT EACH
SUESET IS ASSIGNED TO A GROUF DF SEQUENTIAL LINES.THE ELOCKS OF 9
FOINTS ARE EVALUATEI IN THE NATURAL DRIERING WHERE EACH 3 LINES
WHICH FORM THE BLOCKS ARE TAKEN AT A TIME.

- the arkiy yn will hnlig the coefficient matrix.
- ITER (I) - NUMEER OF ITERATIONS RUN EY FROCESSOR I.
- NFATH = NUMEEF OF FARALLEL F'ATHS .
- $W=$ FELAXATION FACTORS .
- MnXITR = MAXIMUM ITERATIONS.
- EFS = accuracy yalue usen for convergence test.
INTEGEF*2 ITIME
IIIMENSION XN(40, AO), EFR(40,40), ITEF(6), IFLAG(6), ITIME(100)
\$ SHAFEI XN, N,N1,N2,NF,ITEF,IFLNG,NFATH,EFS,MAXITF,W,ITIME
\$USEFRAR
MAXITR $=1000$
$E F S=0.00001$
WFITE ( 6,2 )
FOFMAT ('FROGFAM NAME :- FROGRAM 4.1 "//)
FEAI MESH SIZE, THE NIMEER OF FARALLEL FATHS
AND FEAII W:WI THE RELAXATION FACTORS.
FENT(5,90) N:NFATH,IFRINT,W,W1,WF
FORMAT (I2,1X,I2,1X,12,1X,F5.3,1X,F5.3,1X,F5.3)
$N 2=N-2$
$\mathrm{N} 1: \mathrm{N}-1$
$N F=N 2 / N F$ NTH
IF (NF E ER. 1) NF:
WFITE(6,95) MAXITR:EFSSN2,NFATH

95
c
C
C
0
FUN THE ALGORITHM WITH W WHICH IS INCFEASEI ECCH STEF EY WI IN ORIEF:
TO FINI THE EXACT RELAXATION FACTOR.
$w:=w+w 1$
IF (W, GT, WF) G0 TO 150
WRITE ( 6,864 ) W
864 FOFMAT $(/ 2 Y, W::=F 7,3)$
C
INITIALISE THE COMFONENTS ANI THE EOUNIIARY OF THE MESH.
TM11 I1:-1, N
$\mathrm{XN}(\mathrm{I} 1,1) \div 100.0$
no $22 \mathrm{~J} 1 \because 2, \mathrm{~N}$
YN(I1, J1) $=0.0$
CONTINUE:
CONTINUE
INITIALISE ITEFATION COUNTEFS AND FLAGS FOF ENCH FROCESSOF
[10 $17 \mathrm{I}=1, \mathrm{NF} \cdot \mathrm{ATH}$
ITEF(I) $=0$
IFLAG(I) -1

```
1 7
        CONTINUE
C START TIMING
$FOALLL 1
    E=1./112.
    A:1./224,
    C=1,/16.
    CALL TIMEST
        $FARENII
1
C
C
    $IOFAR 15 IF-1,NFATH
        IF'S-NF'*(IF'-1)+2
        IFF=-NF'*IF+1
C
C ITERATE ON THE SUBSET ELEMENTS UNTIL A CONUERGENCE IS ACHIEVEII.
C
ITEF(IF')-ITER(IF')+1
C FITCK EACH THFEE CONSEQUENT LINES AT A TIME SO THAT THE
C HLOCKS TO EE EVALUATEII CAN EE SOLUEII AS COMFLETE.
C
    IN 35 I=IF'S,IF'F,3
    [10 36 J-2,N2,3
C FTNII THE EQUATIONS OF THE 9 FOINTS OF THE ELOCK.
    XOL[1=XN(I,J)
    R1:-XN(I-1,J)+YN(I,J-1)
    R2=:XN(I-1,J+1)+XN(I+1,.1-1)
    R3:-XN(I-1,J+2)+XN(I,J+3)
    F{:=XN(I+3,J)+XN(I+2,J-1)
    R5:-XN(I+1,J+3)+XN(I+3,J+1)
    F6-XN(I+2,J+3)+XN(I+3,J+2)
    R7:-XN(I+1,J-1)
    F:-XN(I-1,J+1)+XN(I+3,J+1)
    R9=XN(I+1,J+3)
    R10-XN(I+3,J+1)+XN(I+1,J-1)
    Fi:1-XN(I-1,J+1)+XN(I+1,J+3)
    R12-XN(I-1,J+1)
    F13=\NN(I+1,J+3)+XN(I+1,J-1)
    F14=XN(I+3,J+1)
    R15:-K3+F4
    R16=-R1+FA
        R17:F3+F6
        F18-R1+R6
        F19-F1+F3
        F20=R6+RA
        R21=F10+F11
        F22-F18+F15
        XNEW=(67,*F1+22,*R2+7,*R15+6,*R5+3.*F:6)*A
    C
    C EVALUATE THE COMFONENTS WITH W ANII FUTT THE NEW
    C UALUE IN THE ARFFAY XN.
        XNEW=W* (XNEW-XOLII)+YOLII
        ERF(I,J) -AES(XNEW-XOLI)/(1+AES (XOLII))
        XN(I,J)=-XNEW
        XOLI-XN(I+1;J)
        XNEW-(37.*R7+11,*R16+7.*R8+5 **R9+3**R17)*R
```

XNEW：＝W＊（XNEW－XOLII）＋YOLII
ERR（I＋1，J）－－ARS（XNEW－XOLII）／（1＋ヘES（XOLII））
XN（I＋1，J）：$:$ XNEW
XOLII－XN（I＋2，J）
XNEW：（67．＊R14＋22，＊R10＋7，＊F18＋6，＊F11＋3．＊R3）＊A
XNEWに－W＊（XNEW－XOL■）＋YOLI
EFR（I＋2，J）$\because$ ABS（XNEW－XOLII）$/(1+$ NES（YOLII））
$X N(I+2, J)=X N E W$
XOLIT－XN（I，J＋1）
XNEW：$=\left(37, * \mathrm{~F}^{\prime} 12+11, * \mathrm{~F} 19+7, * \mathrm{~F} 13+5, * \mathrm{R} 14+3, * \mathrm{~F} 20\right) * \mathrm{E}$
XNE W：$-W *$（ XNE（J）－XOLII）＋YOLII
EFR（I；J＋1）：－ABS（XNEW－XOLI）／（1＋AES（YOL［I））
XN（I，J +1 ）＝XNEW
XOL［：XN（I＋1，J＋1）
XNE：W：（2，＊R21＋下22）＊C．
XNEW $=W$（ （XNEW－XOLII）+ YOLII
EFF（I $1+1, J+1$ ）＝ABS（XNEW－XOLII）／（1＋AES（XOLII））
XN（I＋1，J＋1）－XNEW
XOL［I－XN（I＋2，J＋1）
XNEW $=(37, *$ R1 $4+11$＊＊R20 7 ，＊F13＋5．＊R12＋3．＊R19）＊E
XNEW：－W＊（YNE
ERF（I＋2：J＋1）$=$ AES（XNEW－XOLII）／（1＋AES（XOLII））
$\times N(I+2, J+1)=$ XNEW
XOLI：－XN（I，J＋2）
XNEW $=(67, * \mathrm{~F} 3+22, * \mathrm{~F} 11+7, * \mathrm{R} 18+6, * \mathrm{~F} 10+3, * \mathrm{~F} 1) * A$

ERF（I $; J+2)$－ARS（XNEW－XOLII）／（1＋AES（YOLII））
XN（I：$J+2)=\mathrm{XNE}$ W
XOLII－XN（I＋1，J＋2）
XNEW：（37，＊R9＋11，＊R17＋7，＊R8＋5，＊R7＋3，＊R16）＊R
XNEW！－W\％（XNE W－XOLI）＋YOLI
EFF $(I+1, J+2)=$ ABS（XNEW－XOL［1）／（1＋AES（XOLII））
XN（I＋1，J＋2）＝XNEW
XOLI：－XN（I＋2，J＋2）
XNEW $=(67, * \mathrm{~F} 6+22, * \mathrm{~F} 5+7, * \mathrm{R} 15+6, * \mathrm{~F} 2+3, * \mathrm{~F} 1) * \mathrm{~A}$
XNE W＝W＊（XNE $\|$－XOLI）＋YOLI
ERR $(I+2, j+2)=A R S(X N E W-X O L I I) /(1+A B S(X O L I))$
XN（I＋2，J＋2）＝XNEW
C
36
35
CONTINUE
CONTINUE
C
CHECK IF THE NUMEER OF ITEFATIONS EXCEELIS
THE ALLOWEI MAYIMUM．
113 IF（ITER（IF），GE，MAYITF）GO TO 15
C
CHECK FOF CONUEFGENCE．

```
NO 777 I1I=IFS,IFF
```

［10 777 J11：2，N1
IF（ERR（II1，J11），GT．EFS）GO TO 50
777
CONTINUE
C
SET THE CONUEFGENT FLAG $=2$ ANLI TEST FOR THE CONUERGENCE OF THE OTHER FROCESSES．

IFLAG（IF）：－2
JJ•1
776 IF（IFLAG（JJ）．EQ．1）GO TO 50
$J コ=J J+1$
IF（JJ ，LE，NF＇ATH）GO TO 776

```
186 15 $FARENL
C
    ENI TIMING
        $IOALL 3
            CALL TIMOUT(ITIME)
3 FOARENH
        WRITE(6,801) ITIME
    801 FORMAT(//'2X,'THE TIMING'/8(I6,2X))
    C
    C CHECK IF ANY F'ROCESS EYCEED ITS MAXIMUM
    C ITERATION LIMITS.
    C
        IO 28 J=1:NFATH
        IF (ITER(J) ,GT, MAXITF) GO TO 250
        CONTINUE
        T1O 85 I=1,NF'ATH
            WFITE(b,812) I,ITER(I)
            FORMAT(//1Y,'CONVERGENCE IS ACHIEVEII IN F'FOCESS NO.'I2,1X,
        1 'AFTEF',IX,IA,IX,'ITEF'ATIONS')
    85 CONTINUE
    154 IF\IFRINT . GT, 1) GO TO 37
        WRITE(6,bA)
    64 FORMNT(//2Y,'THE SOLUTION MESH IS')
        NO 810 IJ:=1,N
            WFITE(6,65)(XN(II,JJ),JJ=1,N)
            FOFMAT(//1Y,7(F10.6,1Y))
65 FOFMAT(
        GO TO 37
        WRITE(6,61)
    250 WRI', FOFMAT(//2X,'NO CONUERGENCE IS ACHIEUEI')
        go T0 15^
    37 GO TO 96
    150 $STOF
        $ENI
```

THIS FROGFAM IMFLEMENTS THE 9-FOINT ELOCK ITEFATIUE METHOI WHEFE
THE SOR ITEFATIUE METHON IS USEII,THIS IS AN ASYNCHRONOUS ALGORITHM
TO SOLVE THE 2-IIIMENTIONAL DIRICHLET FROELEM.THE LINES OF THE
MESH TO EE SOLUED ARE FARTITIONEI INTO NFATH SUESETS SO THAT EACH
SUESET IS ASSIGNEI TO A GROUF OF SEQUENTIAL LINES. THE ELOCKS OF 9
FOINTS ARE EVALUATELIN THE REII-BLACK ORIERING WHERE EACH 3 LINES
WHICH FORM THE BLOCKS ARE TAKEN AT A TIME.
-- THE AFRAY YN WILL HOLIS THE COEFFICIENT MATRIX.

- ITEF(I) =: NUMEER OF ITERATIONS RUN EY FROCESSOR I.
- NFATH =- NUMEEF OF FAFALLLEL FATHS .
- $W=$ RELAXATION FACTORS.
- MAXITK $:=$ MAXIMUM ITERATIONS.
- Ef'S - - accuracy value usen for conveggence test.
INTEGER ITIME,FLAG
IIIMENSION XN(10, 今0), EFF( 40,40 ), ITER(6), IFLAG(6), ITIME(100)
\$SHAREII XN,N,N1,N2,NF,ITER,IFLAG,NPATH,EFS,MAXITF,W,ITIME
\$USEFAR
MAXITR - 1000
EFS $=0.00001$
WFITE (6,2)
FORMAT('FROGRAM NAME :- * FROGRAM 4.2 "//)
FEAII MESH SIZE,NUMEER OF FARALLEL FATHS
ANI W,WI THE RELAXATION FACTORS.
FEALI(5,90) N,NFATH,IFFINT,W,W1, WF
FOFMAT (I2,1X,12,1X,12,1X,F5,3,1X,F5,3,1X,F5,3)
$\mathrm{N} 2=\mathrm{N}-2$
$N 1=N-1$
$N F=N 2 / N F$ NTH
IF (NF' EQ , 1) NF:=3
WRITE(6,95) MAYITR,EFSSN2,NFATH
FOFMAT (/'MAXITF'-', I4/2X,'EFS='F10.6/2X,'N2-', I2/2X,'NF'ATH=', I2)
FUN THE ALGORITHM WITH W WHICH IS INCFEASEI EN:CH STEF EY WI IN ORIEF
TO FINII THE EXACT RELAXATION FACTOF.
$w=w+w 1$
IF (W .GT, WF) GO TO 150
WRITE (6,86!)W
FORMAT (/2Y,'W:=',F7.3)
C
INITIALISE THE COMFONENTS ANI THE EOUNIAFY OF THE MESH.
no 11 I1: $=1, \mathrm{~N}$
$X N(I 1,1)=100.0$
पO $22 \mathrm{~J} 1: 2, \mathrm{~N}$
$\mathrm{Y}_{\mathrm{N}}(\mathrm{I} 1 \mathrm{f}, \mathrm{J1})=0.0$
CONTINUE
continue
INITIALISE ITERATION COUNTERS AND FLAGS FOF EACH FROCESSOR
IO $17 \mathrm{I}=1$ :NFATH
ITER(I)-0
IFLAG(I) $=1$

```
17 CONTINUE
C STOFT TIMING
                                    508
    $HOALL 1
            F=1./112.
            A=1./224.
            C:=1./16.
            CNLL TIMEST
        $F'AREND
C
C
c
    SET UF A FROCESS FOF ENCH FROCESSOR TO ITERATE ASYNCHRONOUSLY.
    $IOF'AR 1S IFM-NF'ATH
        IFS:-NF*(IF-1)+2
        IF'F=NF'*IF+1
    C
C ITERATE ON THE SURSET ELEMENTS UNTIL A CONVERGENCE IS ACHIEVEI.
C
50 ITER(IF')--ITEF(IF')+1
C F FTCK EACH THREE CONSEQUENT LINES AT A TIME SO THAT THE RLOCKS TO EE
    EvAluATED CAN bE SOLVEI AS COMFLETE.
        K1:-5
        FLAG-O
        IO 35 I-IFS,IF'F,3
            K1=7--K1
            HO 36 J=K゙1,N2,6
C
    FINII THE EQUATIONS OF THE 9 FOINTS OF THE ELOCK.
    YOLD=XN(I,J)
        R1:-YN(I-1,J)+YN(I,J-1)
        F2-XN(I-1,J+1)+XN(I+1,J-1)
        F:3:XN(I-1,J+2)+XN(I,J+3)
        R4: XN(I+3;J)+YN(I+2,J-1)
        FE=XN(I+1;J+3)+XN(I+3,J+1)
        R6:XN(I+2,J+3)+XN(I+3,J+2)
        F゙7:XN(I+1,J-1)
        F8-XN(I-1,J+1)+XN(I+3,J+1)
        F9:-XN(I+1,J+3)
        F10:-XN(I+3,J+1)+XN(I+1,J-1)
        F11=XN(I-1;J+1)+XN(I+1,J+工)
        F12=XN(I-1,J+1)
        R13:-XN(I+1,J+3)+XN(I+1,J.-1)
        Fi14-XN(I+3,J+1)
        F15=た3+下㣰
        R16=F1+FM
        R17-R3+R6
        R18-Fi+N'6
        F19=F1+下3
        F20=F66+F1
        R21=R10+F11
        R22-%R18+F15
        XNEW=(67.*R1+22.*F2+7.*R15+6.*R5+3.*R6)*A
    C
    C EVALUATE THE COMFONENTS WITH W ANI FUT THE NE!
    vALUE IN THE ARRAY YN.
        XNEW=W*( XNEW-YOLII)+YOLII
        ERR(I,J)==AES(XNEW-XOLI)/(1+AES(YOL[I))
        XN(I,J)=XNE!!
        YOLI:-XN(I+1,J)
```

C

C
C
C

C

XNEW=W* (XNEW-XOLII) +YOLI
ERF(I+1;J):"ABS (XNEW-XOLI)/(1+AES (XOLII))
XN(I+1,J) $\because$ XNEW
XOL[I=XN(I+2,J)
YNEW $=(67, * \mathrm{~F} 4+22, * \mathrm{~F} 10+7, * \mathrm{~F} 18+6, * \mathrm{R} 11+3$, *F3) *A
XNEW $=W *$ (XNEW-XOLD) + YOLI

XN(I+2,J):-XNEW
XOLII =XN(I, J+1)

XNEW $=W *$ (XNE $!$ - XOLII) +YOLI
EFFi(I, $1+1) \cdots$ AES (XNEW-XOL[I)/(1+ABS (XOLII))
YN(I,J+1)=XNEW
XOLI $=\times$ XN(I+1, $J+1)$
YNEW:- (2 * * $221+\mathrm{F} 22) * C$
XNEW=:W* (XNE $W$-XOLI) + YOLI
EFF (I+1, $1+1$ ) = ABS (XNEW-XOLI) /(1+ABS (XOLII))
$X N(I+1, J+1)=X N E W$
XOLI:-XN(I+2, $1+1)$
XNEW=( 37 * *R14+11,*R20+7**R13+5,*R12+3,*R19)*E
XNEW=W* (XNEW-XOLI) + YOLI
EFF ( $1+2, J+1$ ) =AES (XNEW-XOLII)/(1+AES (YOLII))
XN(I+2, J+1) =XNEW
YOL[1: $\mathrm{XN}(1, J+2)$
YNEW $=\left(67, * \mathrm{~F}^{3} 3+22, * \mathrm{~F} 11+7 \cdot * \mathrm{~F} 18+6, * \mathrm{~F} 10+3 \cdot * \mathrm{~F} 1\right) * \mathrm{~A}$
XNEW=W* (XNEW-XOLI) + YOLI
EFF' (I; J 2 ) $\cdots$ - ABS (XNEW-XOLII)/(1+nES (YOLII))
YN(I, J+2) = XNEW
YOLII-XN(I+1, $1+2$ )
XNEW $=\left(37, * \mathrm{~F}^{\prime} 9+11, * \mathrm{~F} 17+7\right.$, *R8+5.*R7+3.*R16)*E
YNEW…W* (XNEW-XOLII) +YOLI
ERF (I $+1, J+2)=$ AES (XNEW-XOLII) $/(1+$ のES (XOLI) )
XN(I+1, J+2)=XNEW
YOLIV-XN(I+2, J+2)

YNEW $=W *$ (XNEW-XOLII) +YOLI
ERF(It2, $1+2$ ) =AES (XNEW-XOLIV)/(1+ヘES (XOLII))
XN(I+2, $J+2)=X N E W$
36 CONTINUE
35 CONTINUE
IF (FLAG , EQ. 1) GO TO 113
FLAG: $=1$
K1:2
GO TO E1

113 IF(ITER(IF)..GE. MAXITF) GO TO 15
CHECK FOR CONUERGENCE.
[10 777 I11=IFS,IF'F
[10 777 J11:2:N1
IF(ERF(II1,J11) ,GT. EF'S) GO TO 50

```
777 CONTINUE
```

C SET THE CONUERGENT FLAG $=2$ ANI TEST FOR THE
C CONUERGENCE OF THE OTHER FROCESSES.

186
187 188 189 190 191
192 193
194 195
196
197

```
            IFLAG(IF)::2
            JJ=1
776 IF(IFLAG(JJ),EQ. 1) GO TO 50
        JJ=\JJ+1
        IF(JJ .LE. NF'ATH) GO TO 776
    15 $FNRENII
C END TIMING
        $IONLL 3
                CALL TIMOUT(ITIME)
3 $F'AFENL
        WFITE(6,801) ITIME
801 FORMAT(//2Y,'THE TIMING'/8(I6,2Y))
C CHECK IF ANY FROCESS EXCCEII ITS MAXIMUM
C ITEFNTION LIMITS.
NO 28 J-1,NF'NTH
            IF (ITER(J) .GT. MAXITF) GO TO 250
28 CONTINUE
LO 85 I=1:NF'ATH
            WKITE(6,812) I,ITEF(I)
812 FOFMNT(//1Y,'CONUEFGENCE IS ACHIEVEII IN FROCESS NO,'I2,IX,
            1 'AFTEF',1Y,I4,IX,'ITERATIONS')
85 CONTINUE
154 IF(IFRINT .GT, 1) GO TO 37
WFITE(b,bA)
64 FORMAT(//2Y,'THE SOLUTION MESH IS')
IO}810 I, 5=1,
WFITE(6,65)(XN(IJ,JJ),JJ=1,N)
    65 FORMAT(//1X,7(F10.6,1Y))
810 CONTINUE
    GO TO 37
    WFITE(6,61)
    61 FOFMAT(//2X,'NO CONVERGENCE IS ACHIEVEI')
    GO TO 15^
    37 GO TO 96
    150 $STOF
    $ENI
```

THIS FROGRAM IMFLEMENTS THE 9-FOINT ELOCK ITEFATIUE METHON WHERE
C THE SOR ITERATIUE METHON IS USEI.THIS IS AN SYNCHFONOUS ALGORITHM TO SOLUE THE 2-IIIMENTIONAL IIFICHLET FROELEM.THE LINES OF THE MESH TO RE SOLUED ARE FARTITIONEI INTO NFATH SUESETS SO THAT ENCH
C SUESET IS ASSIGNEI TO A GFOUF OF SEQUENTIAL LINES. THE ELOCKS OF 9
C FOINTS AFE EUALUATEI IN THE NATURAL OKIEEING WHEFE EACH 3 LINES
WHICH FORM THE ELOCKS AFE TAKEN AT A TIME.

- the array yn will holis the coeffictent matrix.
- ITER = NUMEER OF ITERATIONS FUN EY THE THE FFOCESSORS.
- NFATH =- NUMEER DF FARALLEL FATHS
- $W=$ RELAXATION FACTORS.
- MAXITR =: MAXIMUM ITERATIONS.
- EF'S = ACCURACY VALUE USEI FOR CONUEFGENCE TEST.

INTEGER*2 ITIME
IIMENSION XN(10,40), ERF (40,40), ITTIME (100)
\$SHAREI XN:N,N1,N2,NF,ITER,IFLAG,NFATH,EFS,MAXITR,W,ITIME
\$USEF'RR
MAXITR $=1000$
EF'S $=0.00001$
WRITE(6,2)
FORMAT('FROGRAM NAME :- FFROGRAM A.3 "///)

ANI WSWI THE RELAXATION FACTORS.
FEALI(5.90) N,NF'ATH,IFFINT,W,W1,WF
FORMAT (12, 1X, 12, 1X, 12, 1X,F5, 3, 1X,F5,3,1X,F5,3)
$\mathrm{N} 2-\mathrm{N}-2$
$N 1=N-1$
$N F=N 2 / N F \cdot \cap T H$
IF (NF , EQ, 1) NF: 3
WRITE (6,95) MAXITR,EFS,N2,NF'ATH
FORMAT (/'MAXITF'-',IA/2X,'EF'S='F10,6/2X,'N2-',I2/2X,'NF'ATH=.', I2)
TO FINE THE EXACT FELAXATION FACTOR.
$w:=W+W 1$
IF (W, GT. WF) GO TO 150
WFITE (6,864)W
864 FORMAT (/2X,'W-.',F7.3)
C
INITIALISE THE COMFONENTS ANI THE GOUNIARY OF THE MESH.
no 11 I1 $=1, \mathrm{~N}$
$X . N(I 1,1)=100,0$
LO 22 J1:-2,N
$\mathrm{XN}(I 1, J 1)=0.0$
CONTINUE
continue
INITIALISE ITERATION COUNTER
ITER $=0$
C
START TIMING
\$IONLL 1

```
        E:=1./112.
        A=1./221.
        C-1./16.
        CALL TIMEST
1 $F'ARENII
C
50 ITER - ITER + 1
C SET UF A FROCESS FOR EACH FROCESSOR TO ITERATE SYNC.HFONOUSLY.
    $LIOF'AR 1S IF=1,NF'ATH
        IF'S:=NF'*(IF-1) +2
        IF'F=NF'*IF'+1
        IO 3S I=IF'S,IF'F,3
        [10 36 J=2,N2,3
C FINL THE EQUATIONS OF THE 9 FOINTS OF THE ELOCK.
    YOLI-XN(I,J)
    F1:-YN(I-1;J)+YN(I;J-1)
    Fi2::XN(I-1;J+1)+XN(I+1,J-1)
    F3:-XN(I-1;J+2)+XN(I,J+3)
    F4=XN(I+3;J)+YN(I+I;J-1)
    R'5=XN(I+1:J+3)+YN(I+3:J+1)
        Fi}6=XN(I+2:J+3)+XN(I+3:J+2
        F7=XN(I+1:J-1)
        F:8:XN(I-1;J+1)+XN(I+3,J+1)
        F9:-XN(I+1,J+3)
        Fi}10=XN(I+3,J+1)+XN(I+1,J-1
        F11=XN(I-1,J+1)+XN(I+1,J+3)
        F12:=XN(I-1,J+1)
        F13:=XN(I+1,J+3)+XN(I+1, J-1)
        R'1A=XN(I+3;J+1)
        た15::Rろ+F゙タ
        K16:-K1+F゙^
        R17:=R3+R゙6
        F18:=R1+N6
        R19:#F1+R゙3
        R20=F:6+Fi4
        R21:-K10+R11
        F22**18+F15
```



```
C
C EVALUATE THE EOMFONENTS WITH W ANII FUT THE NEIN
C UALUE IN THE AFFINY XN.
    XNEW=W*(XNE:W-XOLI) + YOLII
    ERF(I,J)=AES(XNEW-XOLI)/(1+AES(XOLI))
    XN(I,J)=XNEW
    YOLI!:-XN(I+1,J)
    XNEW:*(37.*F7+11,*R1647.*F8+5.*F9+3.*F17)*E
    YNEW=W* (XNEW-XOLI) + YOLI
    EFR(I+1,J)==ABS (XNEW-XOLI!)/(1+AES (XOLII))
    XN(I+1,j):\becauseYNEW
    XOLI:-XN(I+2,\)
    XNEW=(67.*R'4+22.*R10+7.*F:18+6.*R11+3.*R3)*A
    XNEW=W* (XNEW-XOLII) +YOLII
    ERFR(I+2,J)=ABS (XNEW-XOLI)/(1+AES(XOLI))
    XN(I+2;J)-XNEW
    XOLI=XN(I,J+1)
    XNEW=(37.*F12+11.*F19+7**F13+5.*R1A+3.*R20)*E
    XNEW:-W* (XNEW-XOLII)+YOLI
```

XN(I,J+1)-XNEW
XOLI $=$ XN $(I+1 ; J+1)$
XNEW=(2.*F21+下22)*C
XNEW=W* (XNEW-XOL[I) + XOL I
$\operatorname{ERR}(I+1, J+1)=\operatorname{ARS}(X N E W-X O L I) /(1+A B S(X O L I))$
XN(I+1, J+1)=XNEW
XOLI - XN ( $1+2, \mathrm{~J}+1$ )
XNEW $=\left(37, * \mathrm{~F}^{2} 11+11, * \mathrm{~F} 20+7, * \mathrm{~F} 13+5, * \mathrm{R} 12+3, * \mathrm{R} 19\right) * \mathrm{R}$
XNEW=W* (XNEW-XOLII) + YOLL
ERF(I+2,J+1)=AES (XNEW-XOL[I)/(1+ABS (XOLI))
XN(I $+2, J+1)=$ XNEW
XOLI:-XN(I, J+2)
XNEW=(67.*R3+22.*R11+7,*R18+6.*R10+3,*R1)*A
XNEW $=W *($ XNEW $-X O L I I)+$ YOLII
ERF(I, J+2) $=\operatorname{ABS}(X N E W-X O L I) /(1+A E S(X O L D))$
XN(I: $1+2)-X N E W$
$X O L I=X N(I+1, J+2)$

XNEW $=W *($ XNE $(W-X O L I)+$ YOLI
$\operatorname{ERR}(I+1, J+2)=A E S(X N E W-X O L I) /(1+A E S(X O L I))$
XN(I+1,J+2)=XNEW
XOLII $=$ XN(I $+2: J+2$ )
YNEW $=(67, * \mathrm{~F} 6+22, * \mathrm{R} 5+7, * \mathrm{~F} 15+6, * \mathrm{R} 2+3, * \mathrm{R} 1) * \mathrm{~A}$
XNEW $=: W *($ YNEV-XOLII) +YOLII
ERF(I+2, $1+2$ ) =ABS (XNEW-XOLI)/(1+ABS (XOLII)) $X N(I+2: J+2)=X N E W$

## C

36 CONTINUE
CONTINUE
\$FARENI
15
C.

CHECK IF THE ITEFATION AEOVE THE SFECIFICATION LIMIT.

```
113 IF(ITER .GE. MAYITR) GO TO 775
```

c
CHECK FOR CONVERGENCE.
IN 777 II1:-IF'S, IFF
[10 $777 \mathrm{J11:2,N1}$
IF(EFF(I11,J11) .GT. EFS) GO TO 50
777
CONTINUE
C
C ENII TIMING
C
775 \$LOARLL 3
CALL TIMOUT (ITIME)
3 \$FARENI
WFITE (6,801) ITIME
801 FORMAT (//2Y,'THE TIMING'/8(I6.2Y))
C
C CHECK IF ANY FROCESS EXCEEI ITS
C MAXIMUM ITERATION LIMITS.
IF (ITER .GT, MAXITR) GO TO 250
IO $85 \mathrm{I} \div 1$ : NFATH
WRITE(6,812) I,ITER
FORMAT (//1Y,'CONUERGENCE IS ACHIEVEII IN F'ROCESS NO,'I2,1X, 1 'AFTER', IX,I4,IX,'ITERATIONS')
1
'AFTER', $1 \mathrm{X}, \mathrm{I} 4,1 \mathrm{X}$, 'ITERATIONS')
85 CONTINUE
154 IF(IPRINT .GT. 1) GO TO 37
WFITE $(6,6$ A)

| 186 | 64 | FORMAT(//2Y,'THE SOLUTION MESH IS') |
| :---: | :---: | :---: |
| 187 |  | [10 $810 \mathrm{IJ} \mathrm{J}=1, \mathrm{~N}$ |
| 188 |  | WRITE (6,65)(XN(IJ, JJ), JJ= 1 , N ) |
| 189 | 65 | FORMAT(//1X,7(F10.6,1Y)) |
| 190 | 810 | CONTINUE |
| 191 |  | GO TO 37 |
| 192 | 250 | WFITE (6,61) |
| 193 | 61 | FOFMAT(//2X,'NO CONUEFGENCE IS ACHIEUEI') |
| 194 |  | GO TO 151 |
| 195 | 37 | GO TO 96 |
| 196 | 150 | \$STOF |
| 197 |  | SENL |

18664 FORMAT(//2X,'THE SOLUTION MESH IS')
$187 \quad$ In $810 I J=1, N \quad$ WRITE 6,65$)(X N(I J, J J), J J=1, N)$
189 65 FOFMAT (//1X,7(F10.6,1Y))
190810 CONTINUE
191 GO TO 37
192250 WFITE 6,61 )
19361 FOFMAT(//2X,'NO CONVERGENCE IS ACHIEVEI')
$194 \quad$ GO TO 151
$195 \quad 37$ GO TO 96
197
\$ENI

```
C
C
C
THE SOR ITERATIUE METHON IS USEII.THIS IS AN ASYNCHRONOUS ALGORITH
TO SOLUE THE 2-DIMENTIONAL IIJFICHLET FRORLEM.THE LINES OF THE
MESH TO EE SOLUEN ARE FAFTITIONEEN INTO NFATH SUESETS SO THAT EACH
SUESET IS ASSIGNEI TO A GFOUF OF SEQUENTIAL LINES.THE ELOCKS OF 9
ARE EUALUATEII IN THE NATURAL ORDERING WHEFE ENCH 3 LINES WHICH
FOKM THE BLOCKS AFE TAKEN AT A TIME.
- the afRA'Y XN will holis the coefficient matrix.
- ITER(I) - NUMEER OF ITEFATIONS FUN EY FROCESSOR I.
- NFATH = NUMEER OF FARALLEL F'ATHS.
- W = RELAYATION FACTORS.
- MAXITR = MAXIMUM ITERATIONS.
- EFS = ACCUFACY UALUE USEI FOR CONVERGENCE TEST,
    INTEGER*2 ITIME
    IIIMENSION XN(40,40),ERR(40,40), ITER(6),IFLAG(6),ITIME(100)
    $SHAREII XN,N,N1,N2,NF,ITER,IFLAG,NFATH,EFS,MAXITR,W,ITIME
    $USEFPAF
    MAXITR = 2000
    EFS = 0.00001
    WRITE (6,2)
    FORMAT('FROGFAM NAME :- " FROGRAM 4.4 "'//)
    REAI THE MESH SIZE, NUMBEF OF F'ARNLLEL F'ATHS
    ANII W&W1 THE FELAXATION FACTORS.
    FEND(5,90) N,NFATH,IFRINT,W,W1,WF
    FORMAT(I2,1X,I2,1X,I2,1X,F5,3,1X,F5,3,1X,F5,3)
    N2 = N-2
    N1=N-1
    NF=N2/NFOTH
    IF(NF ,ER, 1) NF:-3
    WRITE(6,95) MAXITR,EFS,N2,NPATH
    FOFMAT(/'MAXITR:=',IA/2X,'EF'S='F10.6/2X,'N2-',I2/2X,'NF'ATH=',I2)
C
C
C
96
    W = W+W1
    IF (W .GT. WF) GO TO 150
        WRITE(6,8SA)W
        FGRMAT(/2Y,'W:-',F7.3)
86
C
C
INITIALISE THF COMF'ONENTS ANII THE ROUNIIAFY OF THE MESH.
No 11 I1=1,N
        XN(I1,1):-100.0
        IO 22 J1-2,N
                YN(II,J1)=0.0
            CONTINUE
            CONTINUE
INITIALISE ITEFATION COUNTERS AND FLAGS FOR EACH FROCESSOR
10 17 I=1,NFATH
        ITER(I)=0
        IFLAG(I)=1
```

        CONTINUE
    C START TIMING
\$moall 1
E:=1./112.
CALL TIMEST
\$FARENI
SET UF A FROCESS FOR ENCH FROCESSOF TO ITERATE ASYNCHFONOLISLY.
\$NOFAR 1S IF=1,NF'ATH
IF'S:NF'*(IF'-1)+2
IFF=NF'*IF'+1
C
C ITEFATE ON THE SUESET ELEMENTS UNTIL A CONUERGENCE IS ACHIEUEI.,
C
50 ITEF(IF):=ITER(IF')+1
C F FTCK ENCH THFEE CONSEQUENT LINES AT A TIME SO THAT THE ELOCKS TO EE
C ENALUNTEI CAN EE SOLUEI AS COMFLETE.
NO 35 I=IFS,IFF,3
[0 36 J=2sN2,3
C FINII THE EQUATIONS OF THE 9 FOINTS OF THE ELOCK.
C
F1:=XN(I-1,J)+XN(I,J-1)
F'2=XN(I-1,J+1)
F3
FA:-XN(I+1,J+3)
RS=XN(I+2,J+3)+XN(I+3,J+2)
FG=XN{I+3:J+1)
F7=XN(I+3,J)+XN(I+2,J-1)
F8:=XN(I+1,J-1)
C
S1=R2+F6
52\#F1+F8
53\cdotsK3+R7
S4=R'5+R5+R'5
S5-F1+R7
S6=R3+R33+R3
S7-54+56
S8=R7+R7+R7
S9:=51+S8
S10=R5+F7
S11-R1+R1+R1
S12-56+511
S13:-63+F5
S11:=S8+S11
S15=R1+F3
C
YOLII-XN(I+1,J)
XNEW=(37,*R8+11,*S5+7.*S1+5.*R4+S7)*E
C
C EVALUATE THE COMFONENTS WITH W ANI FUT THE NEIJ
C UALUE IN THE AFR'AY YN,
XNEW=W*(XNE!!-XOLI!)+YOLII
ERR(I+1,J)=AES (XNEW-XOLI)/(1+AES (XOL[I))
XN(I+1,J):-XNEW
XOLII=XN(I,J+1)

```

C

C

C

C
C

C

XNEW＝（37．＊R2＋11，＊S15＋7．＊S2＋5，＊R6＋59）＊E
YNEW：－W＊（XNEW－XOLII）＋YOLI

XN（I，J＋1）－\(=\) XNEW
XOL［：－XN（I＋2，J＋1）
XNEW \(=:(37, * \mathrm{R} 6+11 \cdot * S 10+7 \cdot * S 2+5, * \mathrm{R} 2+512) * \mathrm{~F}\)
XNEW－WW（XNE
ERF＇（I \(+2, J+1)=\operatorname{AES}(\) XNEW－XOLII）\(/(1+\) AES（XOLII）\()\)
XN（I \(+2, J+1)=X N E W\)
YOLIT－XN（I＋1，J＋2）
XNEW \(=(37 \cdot * \mathrm{~F} 1+11 \cdot * 513+7 \cdot * 51+5 \cdot * \mathrm{~F} 8+514) * \mathrm{~B}\)
YNEW：W＊
EFR \((I+1, J+2)=\operatorname{ABS}(\) XNEW－XOLII）／（1＋AES（XOLII））
\(X N(I+1, j+2)=X N E W\)
\(T 2 \cdots X N(I+1 ; J)\)
TA：－XN（I，J＋1）
\(T 6-X N(I+2: J+1)\)
T8：－XN（I＋1， \(\mathrm{J}+2\) ）
XOLI：\(=X N(I+1, J+1)\)
\(X N E W=(T 2+T 4+T 6+T 8) *, 25\)

\(\operatorname{EFR}(I+1, J+1)=\operatorname{ARS}(X N E W-X O L I) /(1+A E S(X O L I))\)
\(X N(I+1, J+1)=X N E W\)
XOLI \(=\) XN（I，J）
YNEW：－（下 \(1+\) T \(2+T 4\) ）＊． 25
XNEW－W＊（XNE！！－XOLI）＋XOLII
EFR（I：J）\(-A E S(X N E W-X O L I) /(1+A E S(X O L[1))\)
XN（I，J）：－XNEW
XOLII＝XN（I＋2，J）
XNEW：（F゙7＋T2＋T6）＊， 25
XNEW \(=W *(\) YNE \(!-X O L D)+\) YOLI
EFF（I＋2，J）：ABS（XNEW－XOLII）／（1＋ABS（XOL［1））
XN（I＋2，J）\(=\) XNEW
XOL［：－XN（I，J＋2）
XNEW：－（F3＋TA＋T8）＊，25
YNEW：＝W＊（YNE \(W\)－XOLD）+ YOLI
EFF（I，J＋2）－AES（XNEW－XOLII）／（1＋AES（XOLLI））
XN（I， \(3+2)=\) XNEW
XOLI \(-\times N(I+2, J+2)\)
XNEW \(=(\mathrm{F} 5+\mathrm{T} 6+\mathrm{T} 8) * .25\)
XNEW＝W＊（XNEW－XOLII）＋XOLII
EFF \((I+2, J+2)=\) AES \((X N E W-X O L \square) /(1+\cap E S(X O L I))\)
\(X N(I+2, J+2)=X N E W\)
36 continue
35 CONTINUE
C CHECK IF THE ITERATION AEOVE THE SFECIFICATION LIMIT．
113 IF（ITEF（IF）．GE，MNXITR）GO TO 15
CHECK FOF CONUEFGENCE．
［10 777 I11 \(=1 \mathrm{IFS}, \mathrm{IFF}\)
10 777 」11：2，N1
IF（ERK（III，J11）．GT．EFS）GO TO 50
777 CONTINUE

C THE CONUERGENCE OF THE OTHER FROCESSORS．
```

186 C
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
2 0 6
207
207
208
209
210
211
212
213
214
215
216
2 1 7
2 1 8
218
220
2 2 1
222
223
2 2 4
225
226
C
IFLAG(IF):-2
JJ-1
776 IF(IFLAG(JJ) .EQ, 1) GO TO 50
JJ!=JJ+1
IF(JJ.LE. NF'ATH) GO TO 776
15 \$F'ARENI
C
C ENII TIMING
C \$IOALL 3
CALL TIMOUT(ITIME)
\$FARENI
WFITE(6,801) ITIME
801 FORMAT(//2X,'THE TIMING'/8(IG,2X))
C CHECK IF ANY FROCESS EYCEENI ITS MAXIMUM
C ITERATION LIMITS.
C
nO 28 J=1:NFATH
IF (ITER(J),GT, MAXITR) GO TO 250
28 CONTINUE
LO 85 I=1,NFNTH
WFITE(6,812) I,ITER(I)
812 FOFMAT(//1Y,'CONUERGENCE IS ACHIEUEII IN F'ROCESS NO.'I2,IX,
812 FOFMAT(//IX,'CONVERGENCE IS ACHIEVE'')
85 CONTINUE
154 IF(IFRINT .GT. 1) GO TO 37
154 IF(IFRINT ;
64 FORMAT(//2Y,'THE SOLUTION MESH IS')
NO 810 IJ:1,N
WFITE (G,6S)(XN(IJ,JJ),JJ=1,N)
65 FORMAT(//IY,7(F10.6,1Y))
810 CONTINUE
GO TO 37
250 WFITE (6,61)
61 FOFIMAT(//2X,'NO CONUERGENCE IS ACHIEVEI')
GO TO 154
37 GO TO 96
150 \$STOF
\$END

```
```

O
C
C
C
C
C

- THE ARFAY YN WILL HOLIIS THE COEFFICIENT MATRIX.
- ITER(I) = NUMEEF OF ITERATIONS RUN BY FROCESSOR I .
- NFATH = NUMEEF OF FARALLEL FNTHS.
- W = FELAXATION FACTORS.
- MAXITF :- MAXIINUM ITERATIONS.
- EPS = ACCURACY UALUE USEF FOR CONUEFGENCE TEST.
INTEGEF ITIME,FLAG
IIIMENSION XN(AO,40),EFF(40,40), ITEF(6),IFLAG(6),ITIME(100)
\$SHAREI XN,N,N1,NZ,NF,ITER,IFLNG,NFATH,EFS,MAXITR,W,ITIME
\$USEF'AR
MnXITF =2000
EFS = 0.00001
WFITE(6,2)
FORMAT ('FROGFANT NAME :- F'ROGRAM 4.5 ! %//)
FENII THE MESH SIZE, NUMEEFR OF F'AFNALLEL PATHS
ANI W:W1 THE FELAXATION FACTOFS.
FENI(5,90) N,NFATH,IFRINT,W,W1,WF
FORMAT(I2,1X,I2,1X,I2,1X,F5,3,1X,F5,3,1X,F5.3)
N2 = N-2
N1=N-1
NF'=N2/NF'\capTH
IF(NF , ER, 1) NF=3
WRITE(6,95) HAXITR,EF'S,N2,NFATH
FOFMAT(/'MAXITR'',I4/2X,'EFS='F10.6/2X,'N2:=',I2/2X,'NFATH:=',I2)
FUN THE ALGORITHM WITH W WHICH IS INCFEASEI ENCH STEF EY WI IN ORLEF
TO FINI THE EXNCT RELAXATION FACTOF.
W=W+WL
IF (W ,GT, WF) GO TO 150
WFITE(6,86A)W
FOFMAT(/2X,'W:"',F7.3)
C
INITIALISE THF. COMFONENTS ANL THE BOLINLAFY OF THE MESH.
IO 11 I1=1,N
XN(II,1)=-100.0
[0 22 J1:-2,N
YN(II,J1):=0.0
CONTINUE
CONTINUE
INITIALISE ITEFATION COUNTEFS ANL FLAGS FOR EACH FROCESSOR
IO 17 I=1,NFATH
ITER(I)=0
IFLAG(I)=1

```
```

17 CONTINUE
C S START TIMING
\$IOALL 1
B-1./112.
CALL TIMEST
1 \$F'ARENL
C S SET UF A FROCESS FOR ENCH FROCESSOR TO ITERATE ASYNCHRONOUSLY.
\$IOF'AR 15 IF=1,NFATH
IFS=NF*(IF'-1)+2
IFF=NF'*IF+1
C ITERATE ON THE SURSET ELEMENTS UNTIL A CONUEFGENCE IS ACHIEVEII,
50
C

```

```

C
FICK EACH THFEE CONSEQUENT LINES AT A TIME SO THAT THE
BLOCKS TO EE EVALUATEI CAN BE SOLVEII AS COMFLETE.
K1 = 5
FLAG =0
IO 35 I=IFSS,IFF,3
K1 = 7-K1
IO 36 J=K゙1,N2,6
C FIND THE EQUATIONS OF THE \& FOINTS OF THE ELOCK.
C
R1:-XN(I-1,J)+XN(I,J-1)
R2=XN(I-1,J+1)
R3: XN(I-1,J+2)+XN(I,J+3)
RA=XN(I+1,J+3)
RG\cdotsXN(I+2,J+3)+XN(I+3,J+2)
F6::XN(I+3;J+1)
R7=XN(I+3,J)+XN(I+2,J-1)
R8=XN(I+1,J-1)
C
51:%R2+R6
S2:=R4+R8
S3=Fに3+R7
S4=R'S+R5+R5
S5-R1+R゙7
S6=R3+R'3+R3
S7=S4+S6
S8=R7+F:7+F7
S9mSA+58
S10=R5+R7
S11-F1+R1+F1
S12=56+511
S13:FR3+F:5
S14=58+511
S15:=R1+R3
C
YOL[I=:XN(I+1,J)
XNEW=(37.*R8+11 * *S +7.*S1+5.*R4+57)*E
C EVALUATE THE COMFONENTS WITH W ANI FUT THE NE!
C EUALUATE THE COM.
XNEW=W*(XNEW-XOLI)+YOLII

```

ERR(I+1,J)=AES(XNEW-XOL[I)/(1+ARS (XOLI))
XN(I+1,J):-XNEU
XOLI-XN(I,J+1)
XNEW=(37.*R2+11•*S15+7.*S2+5.*R6+S9)*E
XNEW=:W* (XNEW-XOLI) +XOLD
ERF(I; \(J+1\) ) =AES (XNEW-XOLII)/(1+ヘES (XOLII))
XN(I, J+1)=XNEW
XOL[I:-XN(I+2,J+1)
XNEW \(=(37 \cdot * \mathrm{~F} 6+11, * S 10+7 \cdot * S 2+5 \cdot * \mathrm{R} 2+512) * \mathrm{E}\)
XNEW=W* (XNEW-XOLI) +XOLI
ERF(I+2: \(1+1\) ) \(=\) AES (XNEW-XOLI)/(1+AES (XOLII))
XN(I+2, \(J+1)=\) XNEW
XOLI:-XN(I+1: \(1+2)\)

XNEW-W - (XNEW-XOLII) +XOLII
EFF(I+1, \(1+2\) ) =AES (XNEW-XOLI)/(1+AES (XOLII))
\(X N(I+1, J+2)=X N E W\)
C
T2:-XN(I+1;J)
\(T A=X N(I, J+1)\)
T6-XN(I+2, \(1+1\) )
T8: \(: \times N(I+1, J+2)\)
C
XOLIM \(\cdots(I+1, j+1)\)
XNEW \(=(T 2+T 4+T 6+T 8) *, 25\)
XNEW \(=W\) * (XNEW-XOLII) + YOLI
\(\operatorname{EFR}(I+1, J+1)=A B S(X N E W-X O L I I) /(1+A E S(X O L I I))\)
XN(I+1, J+1)=XNEW
XOLII-XN(I,J)
YNE: - (R1+T2+TA) *. 25
XNEW-W* (XNE \((\) - - XOLII) + YOLD
ERF(I, J) \(\cdots\) AES (XNEW-XOLII)/(1+ABS (XOLII))
XN(I:J) - XNEW
XOLI=XN(I+2,J)
XNEW=(F7+T2+T6)*. 25
XNEW \(=W\) * (XNE (H-XOLI) + YOLI
ERR(I+2,J)=ARS(XNEW-XOL[I)/(1+AES (XOLII))
XN(I+2,J)=XNEW
\(X O L D=X N(I, J+2)\)
XNEW=(R3+T4+T8)*. 25
XNEW=W* (XNEW-XOLI) +XOLI
ERR(I: \(J+2)=A B S(X N E W-X O L I) /(1+A B S(X O L I))\)
XN(I, J+2)=XNEW
XOLIOXN(I+2, J+2)
XNEW \(=(R S+T 6+T 8) *, 2 E\)
XNEWWW* (XNEW-XOLI) + YOLI
\(\operatorname{ERF}(I+2, J+2)=\operatorname{ABS}(\) XNEW-XOLI) \(/(1+\) ARS (XOLII) )
XN(I+2,J+2)=XNEW
C
36 CONTINUE
35 CONTINUE IF (FLAG .EQ. 1) GD TO 113
FLAG \(=1\)
K1 =2
GO TO 51
C
CHECK IF THE ITERATION AEOUE THE SFECIFICATION LIMIT. 
113 IF(ITER(IF) .GE, MAXITR) GO TO 15
C
CHECK FOR CONUERGENCE.
```

        LO 777 I11=IFS,IFF
            IO 777 J11-2,N1
                IF(ERR(I11,J11) .GT. EF'S) GO TO 50
    777 CONTINUE
C
SET THE CONUERGENCE FLAG
CONUERGENCE OF THE OTHER FROCESSES.
IFLAG(IF'):-2
JJ=1
IF(IFL^G(JJ) .EQ. 1) GO TO 50
J\:=\J+1
IF(JJ +LE, NF'ATH) GO TO 776
\$F'ARENL;
ENLH TIMING
\$IOALL 3
CALL TIMOUT(ITIME)
\$F'ARENEI
WRITE(6,801) ITIME
801 FORMAT(//2X,'THE TIMING'/8(IG,2X))
C
CHECK IF ANY FROCESS EYCEEII ITS MAYIMUM
ITERATION LIMITS.
IO 28 J:=1,NF'ATH
IF (ITER(J) .GT. MAXITF゙) GO TO 250
28 CONTINUE
IO 85 I:-1:NF'NTH
WFITE(6,812) I,ITEF(I)
812 FORMAT(//1X,'CONUEFGENCE IS ACHTEUELI IN FROCESS NO.'I2,1X,
1
'AFTER',IX,I4,1X,'ITERATIONS')
85 CONTINUE
154 IF(IFRINT ,GT. 1) GO TO 37
WRITE (6,64)
FOFMMAT(//IX,'THE SOLUTION MESH IS')
10 810 IJ:-1,N
WRITE(6,65)(XN(IJ,JJ),JJ=1,N)
G5 FORMAT(//1X,7(F10.6,1X))
810 CONTINUE
GO TO 37
250 WRITE(6,61)
G1 FOFMAT(//2X,'NO CONUEFGENCE IS ACHTEVEI')
GO TO 151
37 GO TO 96
150 \$STOP
\$ENI

```
\(C\)
\(C\)
\(C\)
\(C\)
\(C\)
\(C\)
\(C\)
\(C\)
\(C\)
\(C\)
\(C\)
\(C\)
\(C\)
THIS FROGRAM IMFLEMENTS THE 9-FROINT ELOCK ITERATIUE METHOL WHEFE
THE SOR ITERATIUE METHOL IS USEI.THIS IS AN SYNCHRONOUS ALGORITHM
TO SOLVE THE 2-IIIMENTIONAL DIFICHLET FROELEM.THE LINES OF THE
MESH TO EE SOLUEI ARE FARTITIONEI INTO NFATH SUESETS SO THAT EACH
SURSET IS ASSIGNEI TO A GFOUF OF SEQUENTIAL LINES. THE ELOCKS OF 9
ARE EUALUATEI IN THE NATUFAL ORIERING WHERE EACH 3 LINES WHICH
FGKM THE ELOCKS ARE TAKEN AT A TIME.
    - the afRay xn will holiis the coefficient matrix.
    - ITER = MUMEEF OF ITEFATIDNS.
    - NFATH = NUMEEF OF FARNLLEL FATHS.
    - \(W=\) RELAYATION FACTOFS.
    - MAXITF :- MAXIMUM ITEFATIONS.
    - ef's = accurac. y value useg for convergence test.
        INTEGER*2 ITIME
        IIIMENSION XN(AO, 40), ERR(40,40), ITTIME(100)
        \$SHAREII XN:N,N1,N2,NF,ITEF, IFLAG,NFATH,EFS,MAXITR,W,ITIME
        \$USEFAR
        MAXITK \(=2000\)
        EF'S \(=0.00001\)
        WFIT TE 6,2 )
    2 FORMAT('FROGRAM NAME :- FFROGRAM 4.6 " \(/ 1 /\) )
    FEAI THE MESH STZE, NUMEER OF F'AFALLEL F'ATHS
    AND W,WI THE RELAXATION FACTORS.
    REAL(5,90) N,NF'ATH,IFRINT,W,W1,WF
    FORMAT (I2,1X,I2,1X,I2,1X,F5.3,1X,F5.3,1X,F5.3)
    \(\mathrm{N} 2=\mathrm{N}-2\)
    \(\mathrm{N} 1=\mathrm{N}-1\)
        \(N F=N 2 / N F \cdot \cap T H\)
        IF (NF .EQ, 1) NF:=3
        WFITE(6,9E) MAYITR,EF'S,N2,NFATH
        FOKMAT (/'MAXITF'=', IA/2X,'EF'S='F10.6/2X,'N2=',I2/2X,'NFATH=',I2)
        95
        FUN THE ALGORITHM WITH W WHICH IS INCFEASEI EACH STEF EY WI IN ORDER
        TO FIND THE EXACT RELAXATION FACTOR.
        \(W=W+W 1\)
        IF (W.GT. WF) GO TO 150
        WRITE (6,864)W
864 FORMAT (/2Y, 'W:=',F7.3)
C INITIALISE THE COMFONENTS ANII THE EOUNIIARY OF THE MESH.
        IO 11 II \(=1\), N
            \(X N(11,1)=100.0\)
            [1O \(22 \mathrm{~J} 1=2, \mathrm{~N}\)
                \(X N(I 1, J 1)=0.0\)
            Continue
            CONTINUE
        INITIALISE ITERATION COUNTER
            \(I T E R=0\)
C START TIMING
        \$nOALL 1
```

                            B=1./112.
                    CNlL TIMEST
            $F'ARENI
    1
C
SET UF A FROCESS FOR EACH FROCESSOR TO ITERATE ASYNCHRONOUSLY.
ITEF = ITEF+1
\$IOFAR 15 IF:-1,NFOTH
IFS=NF*(IF'-1)+2
IFF=NF'*IF'+1
C
C
C
IN 35 I=IF'S,IFF,3
HO 36 J=2,N2,3
C
C
Fi:=XN(I-1,J)+XNN(I,J-1)
R2=XN(I-1,J+1)
R3:=XN(I-1;J+2)+XN(I,J+3)
R1=XN(I+1,J+3)
RS:XN(I+2,J+3)+XN(I+3,J+2)
F6=XN(I+3,J+1)
F7=XN(I+3,j)+XN(I+2,J-1)
FB=XN(I+1,J-1)
S1=R2+Fi6
S2:%N4+R8
53:-F3+R7
54=R5+R5+RE
S5"*R1+R゙7
56:=R3+R33+R3
57:=5^+56
S8=R7+R7+F7
S9-S4+S8
S10=F5+F7
S11=F1+R1+R1
S12:S6+S11
S13=R3+R5
S1A=58+511
S15=R1+F'3
C
XOL[I:\XN(I+1,J)
XNEW=(37.*F*8+11,*S5+7.*S1+5,*R4+S7)*R
C
C EVALUATE THE COMFONENTS WITH W ANI FUT THE NEN
C value IN THE AFRAY YN.
XNEW=W*(XNEW-XOLD)+YOLI
ERR(I+1,J)=ABS (XNEW-XOLII)/(1+ARS (XOLII))
XN(I+1,J)-XNEW
XOL[I=XN(I,J+1)
XNEW:-(37,*R2+11,*S15+7.*S2+5,*R6+S9)*E
XNEW=W*(XNEW-XOLII)+YOLII
ERR(I;J+1)=AES(XNEW-XOL[I)/(1+AES(XOLII))
XN(I,J+1)=XNEW
XOLD:=XN(I+2,J+1)
XNEW=(37.*R6+11,*S10+7,*S2+5,*R2+S12)*E
XNEW-\cdotsW*(XNEW-XOLI)+XOLI
ERF(I+2,J+1)=ABS(XNEW-XOLI)/(1+तBS(XOLD))

```
\(X N(I+2, J+1)=X N E W\)
XOLI-XN(I+1,J+2)
XNEW \(=\left(37, * \mathrm{R}^{1} 4+11, * S 13+7, * S 1+5 \cdot * \mathrm{~F} 8+514\right) * \mathrm{~F}\)
XNEW-W* (XNEW-XOLII) +XOLI
\(\operatorname{ERF}(I+1, J+2)=\operatorname{AES}(X N E W-X O L I I) /(1+\cap E S(X O L[1))\)
XN(I+1, \(J+2)=\) XNEW
C
T2:-XN(I+1;J)
\(T A-X N(I, J+1)\)
T6: \(\mathrm{XN}(I+2: J+1)\)
T8:-XN(I+1, J+2)
C
XOL \([=: \times N(I+1, J+1)\)
XNEW \(=(T 2+T 4+T 6+T 8) *, 25\)
XNEW:-W\% (XNEW-XOLII) + YOLI
\(\operatorname{EFR}(I+1, J+1)=A B S(X N E W-X O L I) /(1+A E S(X O L I))\)
XN(I+1, J+1)=XNEW
XOLI=XN(I,J)
XNEW: \(=(\mathrm{F} 1+\mathrm{T} 2+\mathrm{T} 4)\) * 25
XNEW \(=W *\) (XNE \(W\)-XOLII) + YOLI
ERR(I,J)-ABS (XNEW-XOLII)/(1+AES (XOLII))
XN(I;J) =XNEW
XOLI \(=\mathrm{XN}(1+2, \mathrm{~J})\)
XNEW:" (F7+T2+T6)*,25
XNEW WW* (XNEW-XOLI) + YOLI
ERF(I+2,J):-AES (XNEW-XOLI)/(1+AES (XOLII))
XN(I+2,J)=XNEW
XOL.I: XN(I,J+2)
XNEW=(R3+T4+T8)*.25
XNEW: =W* (XNEW-XOLII) +XOLI
ERF(I; J+2) :=ABS (XNEW-XOLI)/(1+AES(YOLD))
XN(I, J+2) =XNEW
XOLI:-XN( \(1+2, \mathrm{~J}+2\) )
XNEW \(=(\mathrm{F} 5+T 6+T 8) * .25\)
XNEW \(=W\) (XNEW-XOLI) + YOLI
\(E F R(I+2\lrcorner+2,)=\operatorname{ABS}(X N E W-X O L I I) /(1+A B S(X O L[1))\)
\(X N(I+2, J+2)=X N E W\)
C
36 CONTINJE
CONTINUE
\$FARENRI
15
CHECK IF THE ITERATION AEOUE THE SFECIFICATION LIMIT.
113 IF(ITER .GE, MAYITF) GO TO 775
C
CHECK FOR CONUERGENCE.
C
nO 777 I11:-IFS, IFF
IO 777 J11-2:N1
IF(EFR(II1,J11) .GT. EF'S) 60 TO 50
777 CONTINUE
\(C\)
775
ENOALL 3
CALL TIMOUT(ITIME)
3 \$FARENI
WRITE (6,801) ITIME
801 FOFMAT(//2Y,'THE TIMING//8(I6,2X))
C
CHECK IF ANY FROCESS EXCEEI ITS MAXIMUM
ITERATION LIMITS.
C
```

186
187
188
1 8 9
190
191
192
193
194
195
196
197
198
199
200
2 0 1
202
2 0 3
204
205 3
206 150
2 0 7
LO 28 J=1:NF'ATH
IF (ITER .GT. MNXITR) GO TO 250
CONTINLIE
IO 85 I=1,NF'ATH
WRITE(6,812) I,ITER
812 FORMAT(//1X,'CONUEFGENCE IS ACHIEUEII IN F'ROCESS NO.'I2\&IX,
1
CONTINUE
IF(IFRINT .GT. 1) GO TO 37
WRITE(6,64)
64 FORMAT(//2X,'THE SOLUTION MESH IS')
no 810 IJ JM,N
WRITE(6,65)(XN(IJ,JJ),JJ=1,N)
GE FORMAT(//1Y,7(F10.6,1X))
810 CONTINUE
GO TO }3
250 WRITE (6,61)
61 FORMAT(//2X,'NO CONUERGENCE IS ACHIEVEI')
GO TO 154
GO TO 96
\$STOF
\$ENI

```
C
C
C
C
C
C
C
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
48
49
CALCULATE NUMEER OF CO-OFERATIVE FROCESSORS
    NFROC - 0
    \$TOALL 30
        \$ENTER FEGI
        NFFROC \(\because\) NFROC+1
        \$EXIT FEGI
    \$F'AFENII
    WRITE (t., 1)
    FORMAT (FFOGRAM NAME :- "FROGRAM E.1*'/)
    FERII THE INFIJT FORAMETERS
        REAL(5.2) N, FO,F'S,IFRINT
        FORMAT (I3,1X,F6,3,1X,I2,1X,I1)
        WRITE(6.5)RO
```

            U1(I) = U3(I)
    60 CONTINIE
C GENERNTE F FARALLEL FATHS WHICH FUUS
simultaNEqUSLY tO EUALUATE THE FIFST SWEEF.
\$NOF'AR 250 L-1,F'
IF => IRS(L)
IEND -- IENDS(L)
[O 200 J:IE,IENI
I=2*J-1
IF (J,EQ, 1 ,OR, J ,EQ. N2) GO TO 210
R1 = I1(J)*(E(I)+G(I-1)*U1(I-1)-FETA*U1(I))
F2 = [1:(J)*(E(I+1)-EETA*U1(I+1)+G(I+1)*U1(I+2))
U2(I) - ALF'*R1+G(I)*R2
U2(I+1) :- G(I)*R1+ALF*R2
GO TO 200
IF (J ,EQ, N2) GO TO 220
R1 = I1(1)*(E(1)-EETA*U1(1))
F22 = [1(1)*(B(2)-BETA*U1(2)+G(2)*U1(3))
U2(1) :- ALF**R1+G(1)*F2
U2(2) =G(1)*R1+NLF'*R2
GO TO 200
F1 :- I1(N2)*(F(N-1)+G(N-2)*U1(N-2)-RETA*U1(N-1))
R2 = [11(N2)*(E(N)-BETA*U1(N))
U2(N-1) == ALF'*R1 + G(N-1)*F2
U2(N) - G(N-1)*R1 + ALF*F2
200 CONTINUE
250 \$FNFENL
C
C GENEFATE F FANALLELL FATHS WHICH RUUN
SImultanegusl.y tO Evaluate the secon sweEF.
\$IOFAR 350 L-1,F
IR - IRS(L)
IENI = IENLS(L)
IO 300 J-TR,IENI
I = 2*J
IF (J .EQ. N2) GO TO 310
R1 = I2(J)*(E(I)+G(I-1)*U2(I-1)-EETA*(I2(I))
R2 = II2(J)*(E(I+1)-EETA*U2(I+1)+G(I+1)*U2(I+2!)
U3(I) :- ALF*R1 + G(I)*FR2
ERR(I) = ABS(U3(I) - U1(I))/(1 + AES(UI(I)))
U3(I+1) = G(I)*R1 + ALF*F2
ERR(I+1) :- AES(U3(I+1) - U1(I+1))/(1 + AES(U1(I+1)))
GO TO 300
U3(1) :- (1./ALF)*(E(1)-EET^*U2(1)+G(1)*U2(2))
ERR(1) :- AES(U3(1) - U1(1))/(1 + ABS(U1(1)))
UZ(N) - (1,/ALF)*(E(N)+G(N-1)*U2(N-1)-EETA*U2(N))
ERF(N)= AES(U3(N) - UI(N))/(I+ABS(UI(N)))
300 CONTINUE
\$PARENLI
CHECK IF THE TOTAL. NUMEER OF ITEFATIOS GREATER
THAN THE ALLOWEI MNYIMUM ITERATIONS.
IF (ITER .GT, MAYITR) GO TO 500
C
CHECK FOR CONUERGENCE.
IO 400 I1:=1,N
IF (EFR(I1) .GT. EPS) GO TO 100

```
```

186 400 CONTINUE
187 C
188 500
500 \$NOALL 600
CALL TIMOUT(ITIME)
600
191 C
7 FORMAT\&/'THE TIME TAKEN FOR CONUERGENCE IS'/)
WFITE(6.8)ITIME
8 FORMAT(8(I6,2Y))
196 C
197
198
199 1
200
201
202
203
204
205
206
207
208
209
2 1 0 ~ 1 2
2 1 1
2 1 2
213
214
215
189
\$F'ARENII
192
1937
194
195 8
WFITE(6,11)
11 FORMAT(/'THE NLMERICAL SOLUTION IS'/)
N0 720 I:1,N
U1(I)-F(I*H,FI)
EFF(IT):AES(U1(I)-H13(I))
720 CONTINUE
WFITE(6,3)(U1(I),I-1,N)
WRITE(6,9)
9 FOFMAT(/'THE SOLUTION MATRIX IS'/)
WFITE (6,3)(U3(I),I=1,N)
3 FORMAT(// 7(F10.6.1X))
WRITE(6,12)
12 FORMAT(/'ERROR UALUES ARE'/)
WRITE(6,3)(ERF(I),I=1,N)
700 WFITE(6,1)ITEF
4 FOFMAT(/'CONUEFGENGE IS ACHEIVEI AFTEF ',IA,' IERATIONS'/')
\$stDF
\$ENI

```
```

**** FROGRAM 5.2 ****
THIS FROGFAM IS AN ASYNCHFONOLIS UEFSSION I OF THE FARALLEL A.G.E. METHOLI THAT USEII TO SOLVE FROELEM I (THE LINEAR FROFLEM). IN THIS FAKALLEL UEFSION, IN THE FIFST SWEEF, EACH FROCESSOR EVALUATES ITS FOINTS EY TAKING UF EACH TWO SUCCESSIVE FOINTS AT A TIME STARTING FROM THE FIFST FOINT ANI TERMINATEI AFTER EUALUATING THE LAST TNO FOINTS. WHILE THE SECONI SWEEF TS STAFTEI AFTER THE FIRST SWEEF HAS EEEN COMFILETED. IN THE SECONI SWEEP ENCH FROCESSOF STARTS ITS FROCESSING WITHIN ITS SUESET GY EUALUATING THE FIFST FOINT THEN EACH SUCCESSIUE TWD FOINTS AT A TIME ANLI THE LAST FOINT IS EVALUATEII ON ITS OWN.

- the arkay ul holis the starting values.
- THE AFRAY U2 HOLIIS THE VALUES AFTEF THE SECOND SUEEP.
- the afriay u3 holins the values after the SECONI SWEEP.
- $N$ - SIZE OF INFUT.
- NFROC = NUMEER OF CO-OFERATING FROCESSORS.
- RO :- ACCELERATION FAFAMETER.
- FS - $\quad$ NUMEER OF FARALLEL FATHS.
- IFRINT - USEII TO INIICATE IF FRINTING FEQUIREI.
- ITER = TOTAL NUMEER OF ITEFATIONS.
INTEGEF ITIME (100),N,F,FS,FLAG,ITEF(4)
IIMENSION U1(200), U2(200), U3(200), [11 (100), D2(100)
IIIMENSION FLAG(A),ERF(200), B(200), G(200)
REAL ALF, ALF2,RO, EETA,H,H1
\$SHAREI N, U1, U2, U3,ERR,ITIME,RO,FSS,FLAG,ITER

```
        FK(X1,X2) = 100*((C0S(X1*X2)**2))+2*X2*X2*\operatorname{cos(2*X1*X2)}
```

        FK(X1,X2) = 100*((C0S(X1*X2)**2))+2*X2*X2*\operatorname{cos(2*X1*X2)}
        F(X1,X2)=((EXF(-20.0)/(1+EXF(-20,0)))*EXF(20.0*X1))+
        1 (1/(1+EXF(-20.0)))*EYF(-20.0*Y1)-((COS(X1*X2))**2)
        $USEFAR
        WFITE(6,1)
    FORMAT('F'ROGRAM NAME :- "FROGFRAM S.2"'/)
    REAI THE INFUT F'ARAMETEFS
FEALI(5,2) N,RO,FS,IFRINT
FORMAT(I3;1X,F6,3,1X,I1,1X,I1)
WRITE(6,5)RO
FORMAT(/'W = 'FG6.3/)
WRITE(6,6)N
FORMAT('SIZE OF INFUUT (N) :- ',I3/)
SETTING THE REQUIFEI UARIAELES TO ALL THE FROCESSORS
\$HOALL 211
H=1.0/FLOAT(N+1)
H1 = H*H
FI = 3.141592654
C =200
C1 - 1.0/(1.0+(H1*C))

```
```

            MAYITR = 500
            EFS = 0.000005
            F'-F'S
            GETA =1-RO
            ALF: =: 1+RO
            ALF2 - ALF * ALF'
            N1 = N-1
            N2 = N/2
    C
IN 40 I-1,N1
G(I) = C1
E(I) - -C1*H1*FK(I*H,FI)
U3(I) = 0.0
4O CONTINLIE
C
C
E(1) := --C1*H1*FK(H,FPI)
F(N):- -CI*HI*FK(H*N,FI)
C
NF:= N2/F
NF1 = N/F
C
TO 45 I=1,F
FLAG(I):-1
45 CONTINUE
C
IO 50 I-1,N2
I1(I)-1.0/(ALFF2-(G(2*I-1)**2))
[12(I) - 1.0/(ALF2 - (G(2*I)**2))
5 0 ~ C O N T I N U E ~
211 \$FAREND
C
STORT TIMING
\$riOALL 214
CALL TIMEST
21A \$FAFENI
C
C GENERNTE F F'AFALLEL FATHS WHICH RUNS
C SIMULTANEOUSLY.
\$IOF'AR 450 L-1,F
IE = (L-1)*NF'+1
IENI :-- L*NF'
IF ( L ,EQ. F) IENI:=N2
C
C CALCULATE THE START NND THE ENI
OF ENCH SLIRGFRUF.
IE1 -- (L-1)*NF1+1
IEND1 -- L*NF1
IF (L ,EQ, F) IENII=N'
C
ITER(L) :: 0
100 ITEF(L) =- ITER(L) + 1
IO 60 I=IE1,IENOI
U1(I) - U3(I)
60 CONTINUE
C
C

```
```

    IO 200 J:TIE,IENI
        I-2*J-1
        IF (J.EQ, 1 .OR, J.EQ. N2) GO TO 210
        F1 = II(J)*(B(I)+G(I-1)*UI(I-1)-EETA*(II(I))
        F2 = M1(J)*(E(I+1)-EETA*UI(I+1)+G(I+1)*UI(I+2))
        U2(I) - ALF*R1+G(I)*R2
        U2(I+1)=G(I)*R1+ALF*F2
        G0 TO 200
        IF (J .EQ. N2) GO TO 2.20
        F1 = D1(1)*(E(1)-EETA*U1(1))
        F22= II(1)*(E(2)-EETA*U1(2)+G(2)*U1(3))
        U2(1) = ALF'*R1+G(1)*R2
        U2(2) =G(1)*R1+ALF*R2
        GO TO 200
        F1 = D1(N2)*(B(N-1)+G(N-2)*U1(N-2)-BETA*U1(N-1))
        R2 = [1(N2)*(B(N)-EETA*U1(N))
        U2(N-1) - ALF**R1 + G(N-1)*R2
        U2(N) = G(N-1)*R1 + ALF*R2
    200 CONTINUE
    C
SECONI SWEEF STARTS.
1O 300 J-IE,IENI
I =2*J
IF (J.EG. N2) GO TO 310
F1 - II2(J)*(E(I)+G(I-1)*U2(I-1)-EETA*U2(I))
R2 = [12(J)*(B(I+1)-EETA*U2(I+1)+G(I+1)*U2(I+2))
U3(I) := ALFF*R1 + G(I)*R2
ERF(I) = AES(U3(I) - U1(I))/(1 + AES(U1(I)))
U3(I+1) = G(I)*RI + ALF*R2
ERF(I+1)=: ARS(U3(I+1) - UI(I+1))/(1 + AES(U1(I+1)))
GO TO 300
U3(1) =: (1./ALF)*(E(1)-EETA*U2(1)+G(1)*U2(2))
ERF(1) = ABS(U3(1) - U1(1))/(1 + ABS(U1(1)))
U3(N)=(1./ALF)*(E(N)+G(N-1)*U2(N-1)-BETA*U2(N))
ERF(N)= ABS(UZ(N) - UI(N))/(I+ABS(UI(N)))
CONTINUE

```

```

    CHECK IF NUMRER DF ITERATIONS EXCEEII THE MAYIMUM.
                            IF (ITER(L) ,GT, MNXITF) GO TO 150
    C
CHECK FOR CONUERGENCE.
IO 400 I1:-IB1,IENII
IF (EFK(II) ,GT. EFS) GO TO 100
CONTINUE
C
FLAG(L) - 2
J:1
A30 IF (FLAG(J) .EQ. 1) GO TO 100
J <! J+1
IF (J .LE, F) GO TO 130
C
450 \$FARENI
500 \$IOALL 600
CALL TIMOUT(ITIME)
600 \$FARENI
C
WRITE(6,7)
7 FORMAT(/'THE TIME TAKEN FOR CONUERGENCE IS'/)
WRITE(6,8)ITIME

```
```

186 8 FORMAT(8(IG,2X))
11 FORMAT\&/'THE EXACT SOLUTION IS'/)
IO 720 I:NI,N
UI(I)=F(I*H,FI)
ERR(I)-AES(UI(I)-U3(I))
720 CONTINUE
WFITE(6,3)(U1(I),I-1,N)
WFITE(6,9)
FOFMAT(/'THE SOLUTION MATRIX IS'/)
WFITE(6,3)(U3(I),I-1,N)
FORMMT(// 7(F10.6:1X))
WRITE(6,12)
FOFMAT(/'ERROR UALUES AFE'/)
WFITE(6,3)(EFR(I),I=1,N)
IO 710 I:-1,F
WFITE(6,10)I,ITER(I)
10 CONTINUE
10 FOKMAT</'CONUERGENCE IS ACHIEVEII IN F'ROCESS NO.' I2:1X,
1 'AFTER',IX,IA,IX,'ITERATIONS')
\$STOF
\$ENI

```
のกกดกดดกดกดกดดกด
150
160
    **** FROGRAM 6.1 *****
    THIS FROGRAM IS THE FAFALLEL IMF'LEMENTATION
    OF THE ORIGINAL SHELL SORT ALGOFITHM.
    IN THIS IMFLEMENTATION THE IISTANCE OF
    COMPARISION IS CHOSSEN SUCH THAT :-
        \([1=[N / 2]\) ANH \([(I)=[H(I-1) / 2]\);
    WHERE \(N:-\) SIZE OF INFUT TO EE SORTEII.
    - F' = NUMEER OF GENEFATEI FARALLEL FOTHS.
    - II = IIISTANCE OF COMF'ARISONS.
    THE AR'RAY "INFUT" IS USEI TO STOFEI THE
    GENEFATED UNSORTEI ELEMENTS.
    REAL INFUT (5000), Y
    INTEGER ITIME(100)
    INTEGER I: N, F', [I, K, J
    \$SHAREI ITIME, INFUT, N, I
    \$FEGION REGI
    \$USEFAR
    REAL NUMEEF OF NUMEERS TO EE SORTEI
    WFITE (6.990)
        WFITE (6,995)
        FEALI (5,980) N
        IF (N . LE . 5000) 60 TO 20
        WFITE (6,970)
        GO TO 10
    GENEFATE THE NUMEEFS TO EE SORTEI. THE SUERDUTINE
    FONF (II) WILL GIVE US A FINNIOM NUMEEF EETWEEN THE
    RANGE 0 AND 1 .
        \(\operatorname{In} 30 \mathrm{I}=1, \mathrm{~N}\)
        INFUT(I) = FNNNF(II)
    CONTINUE
        \$IOALL 110
        CALL TIMEST
        \$FARENI
    CNLCULATE THE IISTANCE OF COMFARISION (II),
    [11 \(=[N / 2]\) WHERE \(N\) ENUAL SIZE OF INFUTT
    II(I) : [II(I-1)/2]
    [] MEANS INTEGFAL FARET
        \(n-N / 2\)
        IF (II . LE, O) GO TO 500
        \(F=M-I I\)
        IF (F , GT, II) \(F=\mathrm{I}\)
        \$HOFAR \(400 \mathrm{KO1,F}\)
        I-K
    \(J=I\)
        Y:-INFUT(I + I)
        IF (Y . LT. INFUT (J)) GO TO 190
        INF'UT \((J+I I)=Y\)
\(\mathrm{I}=\mathrm{I}+\mathrm{II}\)
IF ((I+II) .LE. N) GO TO 160
GO TO 400
INFUT(J+II) = INFUT(J)
\(J=\mathrm{J}-\mathrm{It}\)
IF (J.GT. O) GO TO 170
GO TO 180
\$F'ARENII
II -1 I \(/ 2\)
GO TO 140
C
- IIOALL 450
CALL TIMOUT(ITIME)
\$FARENII
C
WFITE THE SORTEII NUMEERS TO THE OUTFUT MEIIA
WFITE(6,940) N
WFITE(6,960) (INFUT(I),I=1,N,50)
C
C
WRITE TIMING
WFITE(6,950)ITIME
\$STOF
FORMAT('SIZE OF INFUT IS \(=\) ',I8/)
FORMAT (8(2Y,I5))
FORMAT (8(2Y,E12.5))
FORMAT('EFROF, YOU EXCEEG MAX. NO., FLEASE TFY AGAIN')
FORMAT (IA)
FORMAT('FROGRAM NAME :-. "FROGRAM 6.1 • '/)
98
\$ENI
```

C
**** FROGFAM 6.2 ****
THIS IS A FAFRALLEL IMFLLEMENTATION OF
THE SHELL SORT ALGORITHM, WHERE THE
IIISTANCE OF COMFARISONS ARE TAKEN AS,
I(1) =- (2**K)+1, I(I) :- (II(I)/2)+1.
WHERE 2**K < N< 2**(K+1),
N = SIZE OF INFUT TO EE SORTEI.
- F' = NUMEER OF GENERATEII FARALLEL FATHS.
- II = IISTONCE OF COMFAFISONS.
THE ARFAY "INPUT" IS USEI TO HOLISS
INFUT ELEMENTS.
FEAL INFUT(5000), Y
INTEGEF ITIME(100), H(14)
INTEGER I: M, F', [I, K', J
\$SHAREII ITIME, INFIUT, N, I
\$FEGION FEG1
\$USEFAF
FEAD NUMEER OF NUMBEFS TO RE SORTEI
WRITE (6,990)
WRITE (6,905)
FEN[I (5,980) N
IF (N .LE, 5000) GO TO 20
WFITE (6,970)
GO TO 10
GENERATE THE NUMEERS TO EE SORTEIF, THE SUEROUTINE
FANF(II) WILL GIVE IS A FANLIOM NUMEER EETNEEN THE
FANGE 0 ANI 1.
IN 30 I=1:N
INFUT(I) = FANF(II)
CONTINUE
\$TIOALL 110
CALL TIMEST
\$FRRENII
CALCULATE THE [IISTANCE OF COMF'AFISION (II),
2~K<N<< 2m(K+1)
H(1) =- (2^K)+1
II(I+1)=(I|(I)/2)+1
K = 1
IF ((2**K) ,GE. N) GO TO 130
K =- k+1
GO TO 120
H(1) - 1
K = バー1
IO 135 J-2,K
H(J)--2**(J-1)+1
CONTINUE
135
C

```
```

110
II = H(K)
538
F=N-[
IF (F .GT. [1) F=IH
\$NOF'AR 400 MT-J,F
I=M
J:=I
Y=INFUT(I+II)
IF (Y .LT. INFUT(J)) GO TO 190
INFUT(J+[I) = Y
I:-I+I
IF ((I+II) .LE, N) GO TO 160
GO TO AOO
INFUT(J+[I) = INFUU(J)
J=J-II
IF (J.GT. O) GO TO 170
GO TO 180
\$F'AFENI
K -- K゙-1
IF (K .LT, 1) G0 TO 500
GO TO 140
C
C
500 \$MOALL 450
CNLL TIMOUT(ITIME)
\$F'AFENII
WRITE THE SORTED NUMEEFS TO THE OUTFIUT MEDIA
WFITE(6,940) N
WFITE(6,960) (INFUT(I),I I 1,N,50)
NKITE TIMING
WFITE(6,950)ITIME
\$STOF
910 FORMAT('SIZE OF INFUT IS = ',IB/)
950 FORMAT(8(2X,I5))
960 FOFMAT(10(2X,E12.5))
970 FORMAT('ERROK, YOU EXCEEII MAX. NO., FIEASE TRY AGAIN')
980 FORMAT(IA)
990 FOFMAT('FROGRAM NAME :- " FROOGRAM 6.2 " '/)
995 FORMAT(, 2^K<N<= 2^(K+1), IM(1)=(2^K)+1, D(I+1):=(IM(I)/2)+1%/)
C
\$ENI
C

```
            **** FROGRAM 6.3 ****
THIS FRGRAM IS THE FARALLEL SHELL SORT WITH
2-WAY MERGE.
THIS FROGRAM FARTITION THE ORIGINAL INFUT SET
OF \(N\) ELEMENTS INTO M SUESET OF SIZE (N/M) EACH.
EACH SUESET THEN SORTEI USING SHELL'S METHOI.
THE SORTEI SUESETS ARE MEFGEI TO FOFM THE FINNL
SORTEII OUTFUT GY USING THE 2-WAY MERGE ALGORITHM,
- INFUT : ARRAY TO HOLIS THE UNSORTEI INFUT ELEMENTS.
- N : NUMBER OF ELEMENTS TO EE SOFTEI.
- NFATH : NUMEER OF FATHS (GROUFS), WHICH IS FOWER OF 2.
- WA : LOCAL ARFAY TO HOLIS THE LOCNL INFUT.
- NFROC : NUMEER OF FROCESSORS USEI.
- NELM : NUMEER OF ELEMENTS IN EACH GROUP.
- INCR : ARRAY TO HOLIS THE INCREMENTS.
- NINCR : NUMEER OF INCREMENTS.
- NF : USEII TO IIETERMINE NUMEER OF FATHS, (NFATH=2**NF).
    INTEGER*2 ITIME
    INTEGER FN: E, E, E1, E1, E2, B2, E11, E22
    IIIMENSION ITIME (100), INCR(1A)
    REAL INFUT(3000), WA (3000)
    COMMON /A/WA
    \$SHAREII INFUT,N,NFATH,NELM,NFROC,MR,FN,INCF,NINCK, ITIME
    \$REGION REGI
    *LISEFAR
CALCULATE NUMEER OF FROCESSORS USEI
    NF'ROC \(=0\)
    \$IOALL 10
        *ENTEF FEG1
            NF'ROC : \(=N F F^{\prime} O C+1\)
        \$EXIT FEGI
    \$FARENII
REAI RIJMEEF OF INFUT ELEMENTS TO EE SOFTETI
    KEAD (S.20) N
    FORMAT (I4)
GENERATE N FANHOM NLIMEEFS
    CALL FANSET(i)
    IOO \(30 \quad \mathrm{I}=1 \mathrm{~N}\)
        INFUT(I) -- FANF (II)
    CONTINUE
    c
    CALCULATE NUMEEF OF FATTHS (GROUF'S), WHICH IS FOWEF OF 2
    RENII(5, AO) NF
    FORMAT (I1)
    NF'ATH = 2**NF'
    NELM \(=\) N/NF'ATH
```

C 540
CALL EUINCR
C \$HOALL 5O
CALL TIMEST
50 \$FARENII
C
\$NOFAF 80 J=1,NF'NTH
E:=NELM*(J-1)+1
E:=NELM*J
IF (J ,EQ, NF'NTH) E =: N
10 60 K?E:E
WA(K)=INFUT(K)
CONTINUE
C
C
CNLL SHELL(E,E)
IO 70 K=E,E
INFUUT(K)=WN(K)
continue
\$FARENI
*IOALL }9
CALL TIMOUT(ITIME)
90 \$FARENI
C
WFITE(6,100)
100 FORMAT(/'THE TIME FOK SORTING'/)
WFITE(6,110) ITIME
110 FORMAT(8(I6,2X))
C
\$HOALL 120
CNLL TIMEST
120 \$FARENI
C THIS FART MEFGE THE SORTEI GROUF'S USING TWO-WAY MERGE
IO 220 MF'=1,NF'
FN=NF'ATH/2**MF
\$NOF'AR 210 J=1,F'N
E1-2**MR*NELM*(J-1)+1
E1=2**(MR-1) *NELM*(2*J-1)
E2=E1+NELM*2**(MRO-1)
E2=E1+NELM*2**(MR-1)
IF (J,EG. FN) E2:WN
E11 = E1
H22= E2
I - E.1-1
I =I+1
IF (INFUT(E11) ,LT. INFUT(E22)) GO TO 1^O
Wr(I) - INFUT(E22)
B22 = E22 + 1
IF (E22 .LE, E2) GO TO 130
GO TO 150
W@(I) = INFUT(E11)
E11 =-111 + 1
IF (E11.LE, E1) GO TO 130
GO TO 170
150 [10 160 K=E11,E1
I=I+1
WA(I):-INFUUT (K)
160 CONTINUE

```
```

                GO TO 190
                HO 180 KM-E22,E2
                        I=I+1
                WCA(I)=INFUT(K)
            CONTINUE
            LO 200 K-E1,E2
                                INFUT(K)-WA(K)
                CONTINUE
                $FAREND
    CONTINUE
        $IOALL
            230
            CALL TIMOUT(ITIME)
    230 \$F'AFENI
WRITE(6,240)
2A0 FORMAT(/'THE TIME FOR MERGING'/)
WRITE(6,110) ITIME
WRITE(6,250) N: NF'NTH, NFFOOC
250 FOKMAT(/10Y,'N =',I8,8X,'NFATHS =',I4,8X,'NFFOC =',IS/)
WFITE(6,260) (INCR(I),I=1,NINCR)
260 FORMAT(/'INCREMENTS',5X,14(I4,2Y)/)
FEALI(5,270) IF'RINT
270 FORMOT (I 1)
IF (IFRINT .NE, 1) GO TO }99
WFITE(6,280)
280 FOFMAT(/'THE SORTED ELEMENTS///)
WRITE(6,290) (INFUT(I),I=1,N)
290 FDFMAT(5(E12,5:2Y)/)
999 \$STOF
\$ENI
C
C THIS SUEROUTINE IEETERMINE THE SERUENCE OF INCREMENTS
GOING TO EE USED EY SHELL SURROUTINE.
SUGROUTINE EUINCR
INTEGEF FN, INCF(14), ITIME:(100)
FEAL INFUT(3000)
\$SHAFEII INFUT,N,NFOTH,NELM,NFFOC,MR,FN,INCF,NINCR,ITIME
I=1
10 INCR(I) = 2**I-1
IF ((2**I) ,GE. NELM) GO TO 20
I = I+1
GO TO 10
20 NINCF=I-1
FETUFN
\$ENII
C
C THIS IS A SHELL'S SORTING SUBROUTINE.
SUBROUTINE SHELL(IE,IE)
INTEGER FN, INCF(14), ITIME(100)
FEAL INFUT(3000), WN(3000)
COMMON /A/WA
\$SHAREII INFUT,N,NFATH,NELM,NFFOC,MK,F'N,INCF,NINCK,ITIME
MTNINCR+1
IO 1O L=1,NINCR
K=M-L
IC-INCR(K)
ICR-IE+IC
IO 30 I=ICR,IE
J:-I-IC
Y=WA(I)

```

187
188 189 190
191
192
193
194
WA(J+IC)=WA(J)
J=J-IC
IF (J.GE, IE) GO TO 10
WA (J I IC) \(=: Y\)
    CONTINUE
    CONTINUE
    RETUFN
    \$ENI
```

C
C
C
THTS IS A F'AFALLEL SHELL SORT METHOL USING THE
OIII. EVEN FEIUCTION MEFGE ALGOFITHM.
THIS FFOGFAM FAFRITION THE ORIGINAL INFUT SET
OF N ELEMENTS INTO M SUBSET OF SIZE (N/M) EACH.
ENCH SUESET THEN SORTEII USING SHELL'S METHON.
THE SOFTEI SURSETS ARE MEFGEII TO FORM THE FINAL
SORTEII OUTFUT USING THE OLL-EVEN REIUCTION.

- INFUT : ARFAY TO HOLIIS THE UNSORTEII INFUT ELEMENTS.
- N : NUMEER OF ELEMENTS TN EE SORTEII.
- NFATH : NLMEER OF FATHS (GROUFS), WHICH IS FOWER OF 2.
- WA : LOCAL ARFAY TO HOLISS THE LOCNL INFUT.
- NFROC : NUMEER OF FROCESSORS USEI,
- NELM : NUMEEF DF ELEYENTS IN EACH GFOUF.
- INCR : ARRAY TO HOLIS THE INCREMENTS.
- NINCF : NUMEER OF INCREMENTS.
- NF: \ USEI TO IIETERMINE NUMEER OF F'ATHS: (NPATH=2**NF).
INTEGEF*2 ITIME
INTEGER FN: Es E, E1, B1, E2, B2; B11% E22
IIMENSION ITIME(100), INCF(11)
FEAL. INFUT(3000), WA(3000)
COMMON /A/WA
C
\$SHAREI INFUT,N,NFATH,NELM,NFROC,MM,MR,FN,INCR,NINCR,ITIME
C
C
C
C
C
NFROC:=0
\$FIONLL 10
\$ENTER FEG1
NFROC = NFROC+1
\$EXIT REG1
\$FAREENL
C
C
FEAII NUMEEF OF INFUT ELEMENTS TO EE SORTEII
FEATI(5:20)N
FORMAT (I4)
GENERATE N FIANLIOM NUMEERS
CALL RANSET(1)
IO 30 I:-1/N
INFUUT(I) =- RNNF(II)
CONTINUE
C
CALCULATE NUMEER OF FATHS (GROUF'S), WHICH IS FOWEF DF 2
REAN(5:40) NF
FORMAT(II)
NF'ATH :- 2**NF'
NELM = N/NF'ATH

```
```

C
CALL E\INCF
C
\$HONLL 50
CALL TIMEST
SO \$FNRENI
C
\$RIOF'AF 80 J=1:NF'NTH
B=NELM*(J-1)+1
E:NNELM*J
IF (J ,EQ. NF'ATH) E :- N
LO }60\textrm{K}=\textrm{E},\textrm{E
WA(K):- INFUT(K)
CONTINUE
CALL SHELL(B,E)
IO 70 K--E,E
INFUT(K) =WA(K)
continue
\$FARENI
\$NOALLL 90
CALL TIMOUT(ITIME)
\$FAFRENE
WFITE(6,100)
100 FORMAT(/'THE TIME FOR SORTING'/)
WFITE(6,110) ITIME
110 FORMAT(B(16,2Y))
sIOALL 120
CALL TIMEST
\$FARENII
C
C THIS FART MERGE THE SORTEII GROUFS USING OILI-EUEN REIUCTION
C AND THO-WAY MERGE
MM = 1
IO 220 MR:=1,NFATH
IF ( MR .EQ. NFATH ,ANI. NF'ATH ,EQ. 2 ) GO TO 22O
IF ( MR .EQ. MM ) GO TO 130
F'N = N'A'ATH/2 - 1
MM =: MM + 2
GO TO 140
130 F'N = NF'ATH/2
140 \$NOFPAF 210 J=1,FN
B1 := 2*NELM*(J-1)+(MM-MF)*NELM+1
E1 =- NFLM*(2*J-1)+(MM-MR)*NELM
E2. - E1+NELM
E2 = E1+NELM
IF ( J.EQ. F'N ) E2=N
E11 = E1
E22 = E2
I = E1-1
I=I + I
IF ( INFUT(B11) ,LT. INFUT(B22) ) GO TO 160
WA(I) = INFUT(R22)
E22 = 822 + 1
IF ( E22 .LE. E2 ) GO TO 150
GO TO 170
WA(I) = INFIJT(E11)

```
```

        E11=E11 + 1
        IF (E11,LE.E1) GOTO 150
        GO TO 185
        LO 180 K - E11,E1
        I=I+1
            WA(I) = INFUT(K)
        CONTINUE
        GO TO 200
        IO 190 K-EN22,E2
            I=I+1
            WA(I) = INFUT(K゙)
            CONTINUE
            HO 20E K゙:E1,E2
            INFUT(K) = WA(K)
            CONTINUE
        $F'AR'ENII
        CONTINUE
        $0OALL 230
            CALL TIMOUT(ITIME)
    230 \$FARFENLI
WRITE(6,240)
FOFMAT(/'THE TIME FOR MERGING'/)
WFITE(6,110) ITIME
WRITE(6,250) N, NFOATH, NFROC
FORMAT (/10X,'N =',IB,8X,'NF'ATHS =', I4,8X,'NF'ROC =',I5/)
WRITE(6,260) (INCR(I),I:=1,NINCR)
FORMAT(/'INCREMENTS',5Y,14(IA,2Y)/)
FEAII(5,270) IF'RINT
FORMAT(I1)
IF (IFRINT ,NE, 1) GO TO 999
WFITE(6,280)
280 FOFMAT (/'THE SOFTEII ELEMENTS'//)
WRITE(6,290) (INFUT(I),I=1,N)
FOFMAT(5(E12.5,2Y)/)
\$STOF
\$ENLI
C
C THIS SURROUTINE DETEFMINE THE SEQUENCE OF INCREMENTS
C GOING TO EE USEN EY SHELL SUEFDUTINE.
C
SURROUTINE EUINCR
INTEGEF FN, INCR(11), ITIME(100)
FEAL INFUT(3000)
\$SHAREIL INF'UT,N,NFNTH,NELM,NFROC,MM,MR,FN,INCF,NINCR,ITIME
I:=1
INCR(I) :- 2**I-1
IF ((2**I) ,GE.NELM) GO TO 2O
I =I I 1
G0 T0 10
NINCF:=I-1
RETURN
\$ENI
C
C THIS IS A SHELL'S SORTING SUEROUTINE,
SUAROUTINE SHELL(IE,IE)
INTEGEF F'N, INCR(11), ITIME(100)
FEAL INPUT(3000), WA(3000)
COMMON /A/WA
\$SHARED INFUT,N,NFATH,NELM,NFROC,MM,MF,FNN,INCR;NINCF,ITIME
M:NINCR+1

```
```

IO 40 L=1,NINCR
546
K:-M-L
IC-INCR(K゙)
ICR=IE+IC
IO 30 I=ICR,IE
J=I-IC
Y:WA(I)
IF (Y .GE, WA(J)) GO TO 20
W^(J+IC)=WN(J)
J:=J-IC
IF (J.GE. IE) GO TO 10
W^(J+IC)=:Y
contiNuE
CONTINUE
FETUFN
SENI

```
```

**** F'ROGRAM 6.5 *****

```
THIS F'ROGFAM IS A FARALLEL EIIGIT SORTING, 1 ist. IMFLEMENTATION.
1. THE N-KEEYS ARE FRESORTEII INTO F-EINS ACCORIING TO THEIF
FIFST I IIIGITS USING SEQUENTIAL AFFFROCH.
2. THE AUAILAKLE FROCESSORS SOKT INTERNALLY THEIR EINS INTO FURSS.
3. THE RUNS AFE GENT SFQUENTIALLY INTO THE COMMON MEMORY WHERE
    THEY CONSTITUTE THE SORTEI LIST.
THE UARIAELES USEI IN THE FROGRAM ARE :-
* INFUT : USER TO HOLT THE INFUT ELEMENTS TO EE SORTELH.
* IEX : USEI TO HOLI THE IIUISORS.
* TAELE : USEI TO HOLI THE NUMEER OF ELEMENTS IN EACH LIST ANI
SUELIST.
* K : USEI TO HOLI THE MIGIYG OF INPUT KEY, WHICH INDICATES
THE LIST AND SUELIST.
* N : NUMEEF OF INFUT ELEMENTS TO EE SORTELI,
* F : THE EACE TiATO WHICH THE INFUT ELEMENTS TO EE CONMEFTEII
* MLIST : HOLIS THE MAIN LIST.
* MLINK : HOLES THE LINK TO THE MAIN LIST.
* SLINK : Hת ItS THE LINK TO THE SUELIST.
* MINIEX : USEI TO CONTFOL THE INIEX OF MAIN LIST.
* Local : usei to holit the infut eil: ents locally.
* \(\because\) ATRIX : USEI TO HOLI THE ELEMENTS OF "TAELE*.
* LIST : USEI TO HOLI THE INIEX OF MNIN LIST.
4. SLIST : USEI TO HOLII THE INIEX DF SUBLIST.
* II : NUMEEF OF IIGITS IKTO WHICH THE INFUT IS EORTEI ON.
        REAL INFUT (102A), LOCAL (1021)
        FENL F'ST, X
        INTEGER *! M1
        INTEGER LIMIT,Y1,Y2,FR.FI
        INTEGER NSS,II,A,E:C,Y:W.Z,COUNT, COUNT1,ITIME(100), START
        INTEGEF IEX(10).TAELE (30,30), K(10)
        INTEGER SLIM世(102A),MINIEX(30),MLINK(102A)
        INTESEE ? 1ATRIX(900), IIST(900), SLIST(900)

        SREGION REG1.
        \$USEF'AR
    CALCULATE NUMEEF OF F'ROCESSORS USEI
        NFFROC=0
        \$ OOALL 10
        \$ENTEK FEG1
            NF'KOC = NFROC 1
        SEXIT REGI
        \$F'ARENI
FEAL NIMREE: QE INFUT ELEMENTS TO BE SORTEI
        F:EAI (5,20) N
```

20 FOFMAT (IN)
C
GENERATE N FIANIIOM NLMMER
I =0
x =-(32767.0 * RANF([I))
I - I+1
INFUT(I) = X
IF (I .LT. N) GO TO 30
C REAII THE GASE TO WHICH THE GASE IS CONVERTEII
C
REAIL (S,40)F
FORMAT(E2,O)
C INTEGEE S IS USEII TO CALCULATE THE IIIUISORS, WHILE IEX HOLII IT
T = ALOG10(2.0)/ALOG10(F)
S = IFIX(16 * T)
M1 = INT(F)
IF ((M1 ** S) \&LE, 32767) GO TO 2
S = S-1
GO TO 1
IEX(1) = M1 ** S
WRITE(6,3)IEY(1),M1,F,S
FORMAT(/2X,'IEX(1) = ',IG,2X,'M1 = ',IA,2X,'FF - ',
1 FG.3:2X,'S = ',I2 /)
C FENII MUMEER OF IIIGIT INTO WHICH THE INFUT IS SORTELI
C
RENII(5,50)II
50 FORMAT(I1)
C
10 60 J:-2,II
IEX(J)=IEX(J-1)/IFIX(F')
60 CONTINUE
C
[10 70 I=1.30
MINIEX(I) =0
70 CONTINUE
C
\$IOALL 75
CALL TIMEST
75 \$FARENI
C SEQUENTIAL FFESORTING OF THE INFUT K゙EYS
E :. IEX(1)
IO 80 Z=1,N
K(1):- INT(INFUT(Z))/E
A = K(1)
R= INT(INFUT(Z))-A*E
K(2) := F/IEX(2)
SLINK(Z) = K(2)
MLINK(Z) -- A
MINNEX(A+1) =- MINIEX(A+1)+1
TABLE(A+1,K(2)+1) =: TABLE(A+1,K(2)+1)+1
80 CONTINUE
C EULIII AN ARROY WITH ALL THE EXISTING SUELISTS IN THE "TAELE*
c=?

```
```

    10 100 I:-1,30
        [10 90 J=1:30
            IF (TAELE(I,J) .EQ. 0) GO T0 90
            c}=c+
            MATRIX(C) -- TAELE(I,J)
            LIST(C) := I-1
            SLIST(C)=J-1
            CONTINUE
    90 CONTINUE
\$LIOALL 105
CNLL TIMOUT(ITISE:)
\$FARENI
WFITE(6:106)
106 FOFMAT(/'TIME FOR IISTRIEUTION'/)
WFITE(6,950) ITIME
FOFMAT(8(16,2Y))
\$NONLL }10
CALL TIMEST
108 \$F'ARENI
C
TRANSFER THE GENEFATED SUELISTS TO THE AUAILARLE FROCESSORS
\$NOFAR 150 Y-1,C
Y1 =Y
W=SLIST(Y1)
COUNT = O
COUNT1 =- 0
STAFT =1
C
calchlate the staft of each sublist
IF (Y .EQ. 1) GO TO 120
Y2 = Y1-1
10 110 M=1,Y2
START = START + MATFIY(M)
CONTINUE
IF (COUNT1 .ER, MATRIX(Y1)) GO TI 145
COUNT1 =: COUNTI+1
COUNT = COUNT+1
IF (SLINK(COUNT), ME, W) GO TO 130
IF (MLINK(COU\&T) .NE. LIET(Y1)) GO TO 130
LOCAL(START) -- INFUT(COUNT)
STAET = START+1
GO TO 120
C
C EALL SORT EURROUTINE
145 LIMIT - START - 1
START = LIMIT - MATRIY(Y1) + 1
IF ( START .ER. LIKIT ) GO TO 150
CALL EUBELE(START,LIMIT)
C
150 \$FNRENI
C
\$IOONLL 160
CALL TIMOUT(ITIME)
\$FARENI
C
WRITE(t:170)

```
```

170 FORWAT(/'TIME FOR SORTING'/)
C
188
18%
100
191
192
193
194
195
196
197
198
1 9 9
200
201
202
203
204
205
20S
207
208
209
210
211
212
213
214
215
2 1 6
217
218
219
220
221
222
223
224
225
226
227
WFITE(6,180)N,NFROC,F'1,C
180 FOFMAT(/5X,'INFUT SIZE = ',I4,EX,'NFROC -- ',I1,2X,'EASE = ',
1 I2,5X,'SUELIST = , I3 /)
C
970 FORMAT(I1)
FEND (5,970) L
IF ( L .NE. 1 ) GO TO 999
WFIITE (6,960)
960 FDFMAT(/'LIST EEFORE SORTING :-'/)
WFITE(6,990)(INFUT(I),I-1,N)
WRITE!\&,980)
980 FORUMNT(/'LIST AFTER SORTING :-'/)
WRITE(6,990)(LOCAL (I),I=1,N)
990 FDRIMNT(A(F17,9,2X)/)
999 \$STOF
\$END
C
C SORT SURFOUTINE USING EURELE SORT TECHNIQUE
C
SUEROUTINE EUEELE (S1,B1)
FEAL INFUT (102A),LOCAL(1024),AREA
INTEGER S1,E1, EOUND,FLAG,ITIME (100)
INTEGER LIST(900),SLIST(900),MATRIX(900),SLINK(1024),MLINK(1021)
\$SHAFEII INFIST,LOCAL,NFFOC,LIST,SLIST,MATRIX,SLINK,MLINK,ITIME
BOUNI - E1 - 1
FLAG = 0
IO 2O I:-S1, BOUNL
IF (LOCAL(I) .LE. LOCAL(I+1)) GO TO 20
AREN :- LOCAL(I)
LOCAL(I) :- LOCAL(I+1)
LOCAL(I+1) = AREA
FLAG :- I
continue
IF (FLNG ,EQ, O) GO TO 30
BOUNLI - FLAG - 1
GO TO 10
3O FETUFN
\$END

```

```

THIS FROGFAM IS A FONFALLEL HIGIT SORTIMG, 2nGS IMFLLEMENTATON.

1. THE N-K゙EYS ARE FRESORTEII INTO F-BINS ACCORIING TO THEIR
FIRST I HIGITS USING FAFALLEL IMFLEMENTATION.
2. THE AVAILAELE FROCESSORS SORT INTERNMLLY THEIR GINS INTO FUNS.
3. THE FUMS AFE SENT SEQUENTIALLY INTO THE COMMON MEMORY WHERE
THEY CONSTITUTE THE SORTEI LIST.
THE UNFIAELES USEI IN THE FROGRAI' ARE :-

* INFUT : USEII TO HOLI THE INFUT ELEMENTS TO EE SORTEII.
* IEX : JSEII TO HOLII THE LIUISORS.
* TABLE : USEII TO HOLII THE NUMBEF OF ELEMENTS IN ENCH LIST ANI
SUBLIST.
* K : USEI TO HOLII THE IIIGITS OF INFUT KEY, WHICH INDICATES
THE LIST ANII SUELIST.
* N : NUMEER OF INFUT ELEMENTS TO RE SORTEIT.
* F : THE EASE INTO WHICH THE INFUT ELEMENTS TO EE CONVERTED.
* MLIST: HOLIS THE MNIN LIST.
* MLINK : HOLIIS THE LINK TO THE MAIN LIST.
* SLINK : HOLIIS THE LINK TO THE SUBLIST.
* MINIEX : USEI TO CONTFROL THE INHEX OF MAIN LIST.
* local : usen to holi the infut elements locally.
* MATFIX : USEI TO HOLII THE ELEMENTS OF "TAELE'.
* LIST : USEI TO HOLII THE THIEX OF MNIN LIST.
* SLIST ; USED TO HOLD THE INLEX OF SUELIST.
* II: NIMEEF OF GIGITS INTO WHICH THE INFUT IS SORTEII ON.
FEALL IMFUT(102^),LOCAL(102^)
FEAL F:T:X
INTEGER*A MI
INTEGEF LIMIT,Y1,Y2,R,E2,E2,IEY1,IEX2
INTEGER N,S,II,A,B,C;Y:W,Z,COURT,COUNTI,ITIME(100),START
INTEGEF IEY(10),TAELE(30,30),N(10)
INTEGER SLINK(1024),MINDEY(30),MLINK゙(1024)
INTEGER MNTTEIY(900),LIST(900),SLIST(900)
\$SHAREI INFUT,ITIME,NFROC,LIST,SLIST,MATFIY,SLINK,MLINK,
1
LOCAL,N,NF'ATH,NELEM,TAELE,MINIEX,IEX
\$REGION FEGI,FEG2
\$USEF'AR
CALCULATE NUMEER OF FROCESSORS USEII
NFROC := 0
\$IONLL 10
\$ENTER FEGI
NFFROC = NFROC+1
\$EXIT FEEGI
\$F'ARENI
FENII NUMEEF OF INFUIT ELEMENTS TO EE SORTED

```
        REAII(5,20) N
20 FORMAT (IA)
    GENERNTE N RANIIOM NUMEER
        I - 0
30 X :- (32767.0 * FinNF(II))
        I=I+1
        INFUT(I) -- X
        IF (I .LT. N) GO TO 30
    C C
C READI THE BASE TO WHICH THE EASE IS CONUERTED
FENI (5,40)F
40 FOFMAT(E2,0)
C IMTEGER S IS USEI TO CALCULATE THE IIIUISORS, WHILE IEX HOLII IT
        T :- ALOG10(2.0)/ALOG10{F'}
        S = IFIX(16 * T)
        M1 =- INT(F)
        IF ((M1 ** S),LE. 32767) GO T0 2
        S:= S-1
        SOTO 1
        IEX(1)=M1**S
    RENI NUMEER OF IIIGIT INTO WHICH THE INFUT IS SORTEII
        FENI(5:50)II
        FORMAT(II)
C
        [0 60 J=2.II
            IEX(J) =- IEX(J-1)/IFIX(F)
        continue
        $NOALL 65
            IEX1 :- IEX(1)
            IEX2 :- IEX(2)
        $FAFENI
        N10 70 I=1,30
            MINIEY(I) = 0
70 CONTINUE
C
        FENTI(5,73)NF'ATH
73 FOFMAT(I3)
        NELEM = N/NF'ATH
C
            $NOALL 75
            CALL TIMEST
    75 $F'AFENI
C THE FARAALLEL FRESORTEI OF THE INFUT KEYS.
        $IOFAR 85 J:-1,NFATH
            E = IEX1
            E2 = NELEM*(J-1)+1
            E2 = N!ELEM*J
            IF (J.ER. NF'NTH ) EZFN
            IO 80 Z - E2,E2
                K(1) -INT(INFUT(Z))/E
            A=k(1)
```

```
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
1A1
142
143
144
145
146
147
118
149
150
151
```



```
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
```

```
            F =INT(INFUT(Z))-A*E
```

            F =INT(INFUT(Z))-A*E
            K(2) =- R/IEX2
            K(2) =- R/IEX2
            $ENTER FEG2
            $ENTER FEG2
            SLINK(Z)=K(2)
            SLINK(Z)=K(2)
            MLINK(Z) = A
            MLINK(Z) = A
            MINIEX(A+1) -: MINIEX(A+1)+1
            MINIEX(A+1) -: MINIEX(A+1)+1
            TAELE(A+1,K(2)+1)= TABLE(A+1,K(2)+1)+1
            TAELE(A+1,K(2)+1)= TABLE(A+1,K(2)+1)+1
            $EXIT REG2
            $EXIT REG2
    80 CONTINUE
80 CONTINUE
8S \$FAFENII
8S \$FAFENII
\$IOALL 86
\$IOALL 86
CNLL TIMOUT(ITIME)
CNLL TIMOUT(ITIME)
86 \$FAFEND
86 \$FAFEND
WFITE (6,87)
WFITE (6,87)
87 FORMAT(/'THE TIME FOF DISTRIEUTION'/)
87 FORMAT(/'THE TIME FOF DISTRIEUTION'/)
WRITE(6,950) ITIME
WRITE(6,950) ITIME
FORMAT(8(I6,2X))
FORMAT(8(I6,2X))
C BULIII AN AFRAY WITH ALL THE EXISTING SUBLISTS IN THE "TABLE*
C BULIII AN AFRAY WITH ALL THE EXISTING SUBLISTS IN THE "TABLE*
C
C
C =0
C =0
DO 100 I=1,30
DO 100 I=1,30
10 90 J=1,30
10 90 J=1,30
IF (TABLE(I;J) .EQ. O) GO TO 90
IF (TABLE(I;J) .EQ. O) GO TO 90
C = C+1
C = C+1
MATRIX(C) = TAELE(I,J)
MATRIX(C) = TAELE(I,J)
LIST(C):" I-1
LIST(C):" I-1
SLIST(C) = J-1
SLIST(C) = J-1
90 CONTINUE
90 CONTINUE
100 CONTINUE
100 CONTINUE
\$DOALL 102
\$DOALL 102
CALL TIMEST
CALL TIMEST
102 \$F'AFENI
102 \$F'AFENI
c
c
TRONSFER THE GENERATEI SUELISTS TO THE AUAILAELE FROCESSORS
TRONSFER THE GENERATEI SUELISTS TO THE AUAILAELE FROCESSORS
\$NOFAR 150 Y-1,C
\$NOFAR 150 Y-1,C
Y1 = Y
Y1 = Y
W=SLIST(Y1)
W=SLIST(Y1)
COUNT = O
COUNT = O
COUNT1 = 0
COUNT1 = 0
START = 1
START = 1
c
c
CALCULATE THE START OF EACH SUELIST
CALCULATE THE START OF EACH SUELIST
IF (Y ,EQ. 1) GO TO 120
IF (Y ,EQ. 1) GO TO 120
Y2 = Y1-1
Y2 = Y1-1
IO 110 M:-1,Y2
IO 110 M:-1,Y2
START = STAFT + MNTFIX(M)
START = STAFT + MNTFIX(M)
110 CONTINUE
110 CONTINUE
120 IF (COUNT1 .ER. MATRIX(Y1)) G0 TO 145
120 IF (COUNT1 .ER. MATRIX(Y1)) G0 TO 145
COUNT1 = COUNT1+1
COUNT1 = COUNT1+1
COUNT = COUNT+1
COUNT = COUNT+1
IF (SLINK(COUNT) ,NE. W) GO TO 130
IF (SLINK(COUNT) ,NE. W) GO TO 130
IF (MLINK(COUNT) .NE. LIST(Y1)) GO TO 130
IF (MLINK(COUNT) .NE. LIST(Y1)) GO TO 130
LOCAL(STAFT) = INFUT(COUNT)
LOCAL(STAFT) = INFUT(COUNT)
START = START+1
START = START+1
GO TO 120
GO TO 120
C
C
C CALL SORT SURFOUTINE

```
C CALL SORT SURFOUTINE
```

```
O
```

O
C
C
C
C
GIUEN A TABLE OF UNOFDEFEII RECORLIS R(1),R(2), * , F(NN),
GIUEN A TABLE OF UNOFDEFEII RECORLIS R(1),R(2), * , F(NN),
WHOSE FESFECTIUE K゙EYS AKE KI,K゙2, ... ,KN. THIS FFRGFAM
WHOSE FESFECTIUE K゙EYS AKE KI,K゙2, ... ,KN. THIS FFRGFAM
SEARCHES FDR A GIUEN ARGUMENT K゙. A IUMMY FECORII R(N+1)
SEARCHES FDR A GIUEN ARGUMENT K゙. A IUMMY FECORII R(N+1)
EEEN INSERTEII AT THE ENII OF THE TAELE TO CONTFOL THE ENTI
EEEN INSERTEII AT THE ENII OF THE TAELE TO CONTFOL THE ENTI
OF THE SENFCH:
OF THE SENFCH:

- N = SIZE OF INFUTT TO BE SEAFCHEII.
- N = SIZE OF INFUTT TO BE SEAFCHEII.
.. NFFOC = NUMEER OF CO-DFERATIUE FROCESSORS.
.. NFFOC = NUMEER OF CO-DFERATIUE FROCESSORS.
- NFATH :- NOMEER OF FAFNLLEL FATHS (SUEGROUFS).
- NFATH :- NOMEER OF FAFNLLEL FATHS (SUEGROUFS).
- NELM }\because\mathrm{ N NIIEER OF ELEMENTS IN EACH SLIBGROUF.
- NELM }\because\mathrm{ N NIIEER OF ELEMENTS IN EACH SLIBGROUF.
- KEYS IS AN ARFNY TO HOLII THE KEYS TO RE SEAFCHEIIFOR.
- KEYS IS AN ARFNY TO HOLII THE KEYS TO RE SEAFCHEIIFOR.
- K = THE AFGUMENT THAT WF. SEARICH FOR.
- K = THE AFGUMENT THAT WF. SEARICH FOR.
FENL INFUT(10000), K; KEYY(6)
FENL INFUT(10000), K; KEYY(6)
INTEGER ITIME(100), FL.AG, E, E, NELM, IFATH(20)
INTEGER ITIME(100), FL.AG, E, E, NELM, IFATH(20)
SHAFEII INFUT, ITIME, N゙, FLAG, N, NFROC, NELM
SHAFEII INFUT, ITIME, N゙, FLAG, N, NFROC, NELM
\$REGION FEGI
\$REGION FEGI
*USEFFAF
*USEFFAF
CALCULATE NUMEER OF FROCESSORS.
CALCULATE NUMEER OF FROCESSORS.
NFROC=0
NFROC=0
\$IOALL 10
\$IOALL 10
\#ENTEF FEG1
\#ENTEF FEG1
NFFOC = NFROC + 1
NFFOC = NFROC + 1
\$EXIT FEG1
\$EXIT FEG1
\#F'ARENII
\#F'ARENII
REAI SIZE OF INF'IT
REAI SIZE OF INF'IT
FENII (5,20) N
FENII (5,20) N
FORMAT (IE)
FORMAT (IE)
GENERATE AND STORE RANLOM NUMEEFS
GENERATE AND STORE RANLOM NUMEEFS
In 40 I =1,N
In 40 I =1,N
INFUT(I) - FNANF(I)
INFUT(I) - FNANF(I)
CONTINUE
CONTINUE
GENERATEI NUMEENS, FRINTEII OF NOT
GENERATEI NUMEENS, FRINTEII OF NOT
FEAI(5,30) IFFFNT
FEAI(5,30) IFFFNT
FOFMAT(II)
FOFMAT(II)
C
C
REAL IN NIMEEF OF FATHS (NFATH) DR SUEGRDUF'S
REAL IN NIMEEF OF FATHS (NFATH) DR SUEGRDUF'S
IF=0
IF=0
IP=IF + I
IP=IF + I
REALI(5,5G) IFATH(IF')
REALI(5,5G) IFATH(IF')
IF (IF'^TH(IF') ,NE. 999) GO TO 50
IF (IF'^TH(IF') ,NE. 999) GO TO 50
55゙ FOFMAT(IJ)
55゙ FOFMAT(IJ)
IF=IF-1
IF=IF-1
C

```
C
```

```
    C FEAII THE ARGUMENT K TO EE SEARCHEII FOF
    C
60 KE K = N = KE % 1
        KENI\(5,65)K゙EYS(K゙E)
        IF (KEYS(N゙E) .NE. .11111111) GO TO60
65 FORMAT(F11.8)
        K゙E = K゙E-1
        IO 888 IFF:-1, TF
            NF'ATH =- IF'ATH(IF'F')
            LOTO7 K゙EE=1.K゙E
                    K = KEYS(KEE)
C
    CALCULATE NUMEEF OF ELEMENTS IN ENCH SIIEGROUPS
    NELM - N/NF'ATH
        $IIOALL 70
            CNLL TIMEST
        $F'ARENLI
C
    FLAG=0
C GENEFIATE "NFATH" FONANLEL FONTHS THAT FUN
C SIMULTANEOUSLY.
    $FIOFAR 7S J=1:NFATH
        IF ( FLAG.EQ, 1 ) GO TO 7%
        E = NELM* (J-1)+1
        E E NELM*J
        IF (J,EQ, NFATH)EN
        CALL SEAFCH (E,E)
        $F'ARENII
        $HOALL EO
            CALL TIMOUT(ITIME)
80 $F'AFENL
C
    IF (FLAG EQ. 1) GO TO 160
C
    WRITE (6,120) K
120 FORMAT (/'THE ARGUMENT ',F11.8,' NOT FOUNL'/)
    WRITE ( 6,1^O)
140 FOFMAT('THE FROGRAM TEFKMINATEF UNSUCCESSFULLY'/)
    G0 TO 200
160 WFITE(6,180) K゙
180 FOFMAT(/'THE AFGUMENT ',F11.8,' FMUNI'/)
    WRITE(6,190)
    190 FORMAT('THE FROGRAM TERMINNTEI SUCCESSFULLY')
    C
200 IF (IFRNT .NE. 1 ) GO TO 240
    WFITE(6,2つ0)
220 FOFMMAT(/'THE UNOFIEFELI ELEMENTS OF THE TAELE ARE :-'/)
    WFITE(6,230) (INFUT(I),I-1,N,50)
    FOFRMAT(S(F11.,8;3X)/)
C
2AO WRITE(6,2FO)
2\XiO FOFMAT(/'THE TIME FOR SENRCHING'/)
    WRITE(G,2AO) ITIME
260 FORMAT (S<I6,2X))
    WKITE(6,270) N, NFROC, NFATH
270 FOFMAT (/2X,'INFUT SIZE = 'IS,SX,'NO, OF FROC. = ',14/
```

```
124
125 777
CONTINUE
126 888 CONTINUE
127 $STOF
128 SENI
129
C SEQUENTIAL SEARCH ALGORITHM
C
SUBROUTINE SENFCH (E1,E1)
FEAL INFUT(10000), K, KEYS(6)
INTEGER ITIME(100), FLAG, B, E, E1, E1, NELM, IFATH(20)
$SHAREII INFUT, ITIME, K, FLAG, N, NFFROC, NELM
I = E1
IF (K .ER, INFUT(I) ) GO TO 30
I = I+1
IF (I .GT. ET ) GOTO 40
GO TO 10
142 40
FLAG = 1
FETURN
$ENI
```

```
C
C
    **** FROGRNM 6.8 ****
    THIS FROGRAM IS THE FARALLEL IMFLEMENTATION OF
    EINARY SEARCH ALGORITHM.
    GIVEN A TAELE OF FECORDS R(1),F(2), ... ,R(N),
    WHOSE KEYS AFE IN INCREASING ORIER
    K゙1 & K2 & ... & KN. THIS FROGRAM SEARCHES FOR
    A GIUEN ARGUMENT K. TO START BY COMFARING K TO
    THE MIIIILE KEY IN THE TAELE, THE RESULT OF THIS
    FROEE TELLS WHICH HALF DF THE TAELE SHOULII BE
    SEARCHEI NEXT, ANI THE SAME FRDCEDURE CAN GE USED
    AGAIN, COMFARING K TO THE MIIINLE KEY OF THE
    SELECTEI HALF, ETC. THIS IS KNOWN AS 'BINAFY SEARCH*.
    - INFUT IS AN ARRAY HOLDS THE INFUJT ELEMENTS.
    - NFROC = NUMEER OF CO-OFEFATIVE FROCESSORS.
    - NFATH =% NIMMEER OF F'ARARLLEL F'ATHS.
    - NELM = NUMEER OF ELEMENTS IN EACH FATH.
    - K = THE ARGUMENT WE SEARCH FOR.
    - N = SIZE OF INFUT ELEMENTS.
    - KEYS IS AN AFRAY THAT HOLIIS THE SEARCHED KEYS.
        FEAL INFUT(9500), K, KEYS(6)
        INTEGER ITIME(100), FLAG, L, U, IFATH(20)
        $EHAREII INFUT, ITIME, K, FLAG, N, NFFOC, NELM, NFATH
        $FEGION REG1
        $USEF'AR
    C
C
C
CALCULATE NUMEER DF F'ROCESSOK
        NPFOC = 0
        $NOALL }1
        $ENTER REG1
            NFROC = NF'ROC + 1
        $EXIT REG1
    $FARENI
    READ SIZE OF INFUIT
        REAI (5,20) N
        FORMAT (IA)
    GENERATE ANII STORE RANIOM NUMEERS
        LO 40 I =1,N
            INFUT(I) = RANF(II)
        CONTINUE
        40
C
```



```
    FRINT THE GENERATEI NUMEERS OR NOT ?
        READ(E,45) IFRNT
        FORMAT(II)
45
C
SORT THE GENEFATED INFUT RANIOM NUMEEFS
USING SHELL'S METHOI
CALL SHELL
C
```

```
C REAII IN NUMBEF OF F'ATHS (NF'ATH) OF SURGROUF'S
C
5 0
55
C
C REAI THE ARGUMENT K TO EE SEARCHED FOR
    KE =O
    KE = KE+1
    REAII(5,65) KEYS(K'E)
        IF (KEYS(KE).NE, ,111111111) GO TO 60
        FORMAT(F14.11)
        KE =KE-1
        [10 888 IFF=1,IF
        NF'ATH = IF'ATH(IFF)
        LO }777\textrm{KEE:=1,K゙E
            K = KEYS(KEE)
    C
C CALCULATE NUMEER OF ELEMENTS IN EACH SURGROUFS
    NELM := N/NF'ATH
C
    $IOALL }7
        CALL TIMEST
        $FARENI
    C
        n0 90 MM = 1,50
        FLAG == 0
    C G GENERNTE "NFOTH" THAT FIUNS SIMULTANEOUSLY.
        $DOF'AF 7S J-I:NFATH
        IF ( FLAG .EQ, 1 ) GO TO 75
        L = NELM * (J-1) + 1
        U = NELM * J
        IF ( J .ER. NF'ATH ) U:=N
        CALL SEAFCH (L,U)
        $FARENI
    C
    90 CONTINUE
        $DOALL 80
        CALL TIMOUT(ITIME)
    80 $FAREND
    C
        IF ( FLAG ,EQ. 1 ) GO TO 160
    C
        WRITE (6,120) K
        FORMAT (/'THE ARGUMENT ',F14.11,' NOT FOUNI'/)
        WRITE(6,140)
    140 FORMAT('THE FROGRAM TERMINATEII UNSUICCESSFULLY'/)
        GO TO 200
        WRITE(6,180) K
    180 FORMAT(/'THE ARGUMENT ',F14.11,' FOUND'/)
        WRITE(6,190)
        FORMAT('THE FROGRAM TERMINATEN SUCCESSFULLY')
    8
C
```

```
200 IF (IFRNT .NE. 1 ) GO TO 240
        WRITE(6,220)
220 FORMAT(/'THE ORDEREI ELEMENTS OF THE TARLE ARE :-'/)
        WFITE(6,230) (INFUT(I),I=1,N,SO)
230 FORMAT(S(F1A.11,2Y)/)
C
240 WRITE(6,250)
250 FOKMAT(/'THE TIME FOR SENRCHING'/)
        WRITE(6,260) ITIME
260 FOKMAT(8(Ib,2Y))
        WRITE(b,270) N, NFROC, NFATH
        FORMAT(/2Y,'INFUUT SIZE = 'IA,SX,'NO, OF F'ROC, = ',IA/
        C /, NUMEEF OF FATHS (SURGROUF'S) =" ',I3/)
            CONTINUE
    888 CONTINSJE
        #Stof
        $ENI
C
C EINAFY SEARCH ALGORITHM
        SURFOUTINE SEARCH (LI,U1)
        REAL INFUT(9500), K, KEYS(6)
        INTEGER ITIME(100),FLAG,L,U,NELM,L1,U1,L2,U2,IFATH(20)
        $SHAFEI INFUT, ITIME, K, FLAG, N, NFROC, NELM, NFATH
        L2 == L.1
        U2 = U1
        IF ( U2 .LT. L2 ) Gח TO 70
        I= (L2+U2)/2
        IF ( K .LT. INFUT(I) ) GO TO 40
        IF (K.GT. INFUT(I) ) GO TO 50
        FLAG == 1
        GO T0 70
```



```
C AIIJUST U2
40 U2 = I-1
        GO TO 10
C ALJJUST L2
        L2 = I +1
        GO TO 10
        RETUFN
        $END
    SHELL'S SORTING SUEROUTINE
        SUBROUTINE SHELL
        REAL INFUT(9500), K, Y, KEYS(6)
        INTEGER ITIME(100), FLAG, I, I, IF'ATH(20)
        $SHAREII INFUT, ITIME, K, FLAG, N, NF'ROC, NELM, NF'ATH
C
    cnlculate the mistance of comfarigion
        I:1
        IF ((2**I) ,GF.N) GO TO 3O
        I=I+1
        GO TO 20
        I=(2**(I-1)-1)
C
        IF ( II .LE. O ) GO TO 100
        I =1
```

```
186 50 J=I
187 Y = INFUT(I+II)
188 60
18970
    IF ( Y , LT. INFUT(J) ) GO TO 80
    INFUT(J+D) = Y
        I =: I+1
        IF ((I+I) .LE, N) GO TO 50
        I}=(\textrm{I}-1)/
        GO TO 40
    80 INFUT(J+[I) =- INFUT(J)
        J=J-I
        IF ( J .GT. O ) GO TO 60
        GOTO}7
        100
        RETUFN
        $ENI
```


## REFERENCES

Abel, et al [1969]: "TRANQUIL: Language for an Array Processing Computer", Proc.Spring Joint Computer Conference, 1969, pp.57-68.

Ackerman, W. and Dennis, J.B. [1979]: "VAL - A Value-Oriented Algorithmic Language Freliminary Reference Manual", Technical Report 218, MIT Laboratory for Computer Science, June 1979.

Adams, D.A. [1970]: "A Model for Parallel Computations", In Parallel Processor Systems, Technologies and Applications, ed. by L.C. Hobbs et al., Spartan Books, Washington, pp. 311-333.

Aho, et al. [1974]: Aho, A.V., Hopcroft, J.E. and Ullman, J.D.: "The Design and Analysis of Computer Algorithms", Addison-Wesley, Reading, Massachusetts, 1974.

Anderson, G.A. and Jensen, E.D. [1975]: "Computer Interconnection Structures: Taxonomy, Characteristics and Complexes", Computing Surveys, Vol. 7, No. 4, pp. 197-213.

Anderson, James P. [1965]: "Program Structure for ParalleZ Processing",: CACM, Vol. 8, No. 12, pp. 786-788.

Arms, et al. [1956]: Arms, R.J., Gates, L.D. and Zondek, B.: "A Method of Block Iteration", J.Soc.Indust.Appl.Math., vol. 4, pp. 220-229.

Baase, S. [1978]: "Computer Algorithms: Introduction to Design and Analysis", Addison-Wesley, Reading, Massachusetts, 1978.

Baer, J.L. and Bovet, D.P. [1968]: "Compilation of Arithmetic Expressions for Parallel Computations", Proc. IFIP Congress 1968, North-Holland, Amsterdam, pp. 340-346.

Baer, et al. [1970]: Baer, J.L., Bovet, D.P. and Estrin, G.:
"Legality and Other Properties of Graph Models of Computations", J.Ass.Comput.Mach., Vol. 17, pp. 543-552.

Baer, J.L. [1973]: "A Survey of Some Theoretical Aspects of Multiprogramming", Computing Surveys, Vol. 5, March 1973.

Baer, J.L. [1982]: "Techniques to Exploit ParalleZism", In Parallel
Processing Systems, An Advance Course, ed. D.J. Evans, Cambridge University Press, 1982, London, pp. 75-99.

Barlow, R.H. and Evans, D.J. [1977]: "An Analysis of the Performance of a Dual-Minicomputer Parallel Computer System", Internal Report No. 59, Computer Studies Dept., Loughborough University.

Barlow, et al. [1981]: Barlow, R.H., Evans, D.J., Newman, I.A. and Woodward, M.C.: "A Guide to Using the NEPTUNE Parallel Processing System", Internal Report, Computer Studies Dept., Loughborough University.

Barlow, et al. [1982a]: Barlow, R.H., Evans, D.J. and Shanehchi, J.: "Performance Analysis of Algorithms on Asynchronous Parallel Processors", Computer Physics Communications, 26, pp. 233-236.

Barlow, et al. [1982b]: Barlow, R.H., Evans, D.J., Newman, I.A., Shanehchi, J. and Woodward, M.C.: "Performance Analysis of Parallel Algorithms on Asynchronous Parallel Computers", Internal Report 157, Computer Stuđies Dept., Loughborough University.

Barlow, R.H. and Evans, D.J. [1982]: "Parallel Algorithms for the Iterative Solution to Linear Systems", The Computer Journal, Vol. 25, No. 1, pp. 56-60.

Barnes, J.G.P. [1984]: "Programming in Ada", Addison-Wesley, Reading, Massachusetts, 2nd ed. (1984).

Batcher, K.E. [1968]: "Sorting Networks and Their Applications", Proc. AFIP Spring Joint Computer Conference, Vol. 32, pp. 307-314.

Baudet, G.M. [1978]: "Asynchronous Iterative Methods for Multiprocessors", J.ACM, Vol. 25, No. 2, pp. 226-244.

Baudet, G.M. and Stevenson, D. [1978]: "Optimal Sorting Algorithms for Parallel Computers", IEEE Trans. on Computers, Vol. C-27, No. 1, pp. 84-87.

Baudet, G.M. [1981]: "Parallel Algorithm: Design, Analysis and Experiments", INRIA, Domaine de Voluceau, 78150, Le Chesnay, France.

Bell, C.G. and Newel1, A. [1971]: "Computer Structures: Readings and Examples", McGraw-Hill, N.Y. 1971.

Bell, W.W. [1975]: "Matrices for Scientists and Engineers", Van Nostrand Reinhold Company, 1975.

Bentley, J.L. and Kung, H.T. [1979]: "A Tree Machine for Searching Problems", Proceedings 1979 International Conference in Parallel Processing, pp. 257-265, August, 1979.

Bernstein, A.J. [1966]: "Analysis of Programs for Parallel Processing", IEEE Trans. on Electronic Computers, Vol. EC-15, No. 5, Oct. 1966, pp. 757-763.

Biggins, M.J. [1980]: "The Numerical Solution of Elliptic Partial Differential Equations by Finite-Difference Methods", Ph.D. Thesis, Computer Studies Dept., Loughborough University.

Birkhoffet al [1962]: Birkhoff, G., Varga, R.S. and Young, D.M.: "Alternating Direction Implicit Methods", Advances in Computers, Vol. 3, Academic Press, New York.

Bustand, D.W. [1980]: "An Introduction to PASCAL-PLUS", in 'On the Construction of Programs', ed. by R.M. McKeag and A.M. MacNaugthen, Cambridge University Press, 1980.

Carre, B.A. [1961]: "The Determination of the Optimal Acceleration Factor for Successive Over-Relaxation", Compt. Journal, Vol. 4, pp. 73-78.

Chazan D. and Miranker, W. [1969]: "Chaotic Relaxation", in Linear Algebra and Its Applications, Vol. 2, pp. 199-222.

Chen, S.C. and Kuck, D.J. [1975]: "Time and Parallel Bounds for Linear Recurrence Systems", IEEE Trans. on Comp., Vol. C-24, pp.701-717.

Chen, T.C. [1975]: "Overlap and Pipeline Processing", In Introduction to Computer Architecture, H.S. Stone, ed. p.p. 375-431, Science Research Associates, Chicago, Illinois.

Clint, M. and Perrott, R.H. [1980]: "Algomithms for Parallel Computers", In On the Construction of Programs, ed. McKeag, R.M. and MacNaugthen, A.M., Cambridge University Press, pp. 283-318.

Commer et al [1971]: Commer, F., Holt, A.W., Evan, S. Dalleli, A., "Marked Directed Graph", J. Comp.Sys.Sci., Vol. Oct. 1971.

Colvin, A. [1980]: "Parallel Processing with Minicom Communications, Vol. 3, No. 3.

Conrad, V. and Wallach, Y. [1977]: "Iterative Solution of Linear Equations on a Parallel Processor System", IEEE Trans. on Computers, Vol. C-26, No. 9, Sep. 1977.

Conte, S.D. and de Boor, C. [1982]: "Elementary Numerical Analysis", McGraw-Hill Book Company, New York.

Conway, M.E. [1963]: "A Multiprocessor System Design", Proceeding Fall Joint Computer Conference.
[Cray 75]: "The CRAY-1 Computer", Preliminary Reference Manual, Cray Research Inc., Chippewa Falls, Wisc. 1975.

Dennis, J.B. and Van Horn, E.C.: "Programming Semantics for Multiprogrammed Computations", CACM, Vol. 9, No. 3, pp. 143-155.

Dennis, J.B. and Misunas, D.P. [1974]: "A Computer Architecture for Highly Parallel Signal Processing", Proc. Annual Conf. CAM, pp. 402-409.

Dennis, J.B. and Misunas, D.P. [1975]: "A Preliminary Architecture for a Basic Data-Flow Processor", Proc. 2nd Annual Symp. on Computer Architecture, IEEE, New York, pp. 126-132.

Dew, et al. [1983]: Dew, P.M., Buckley, T.F. and Berzins,M.: "Application of VLSI Devices to Computational Problems in the Gas Industry", 163 Dept. of Computer Studies, Leeds University.

Dew, P.M. [1984]: "VLSI Architecture for Problems in Numerical Computation", In Supercomputers and Parallel Computation, edited by Paddon, D.J., Clarendon Press, Oxford.

Dijkstra, E.W. [1968]: "Co-operating Sequential Process", In Programming Languages, edited by Genuys, F., Academic Press, pp. 43-112.

Dunbar, R.C. [1978]: "Analysis and Design of Parallel Algorithms", Ph.D. Thesis, Computer Studies Dept., Loughborough University.

Enslow, P.H. [1974]: "Multiprocessors and Parallel Processing", edited by Enslow, P.H., A Wiley-Interscience Publication, John Wiley and Sons, New York.

Enslow, P.H. [1977]: "Multiprocessor Organization - A Survey", Computing Surveys, Vol. 9, No. 1, pp. 103-129.

Evans, D.J. and Williams S.A. [1978]: "Analysis and Detection of Parallel Processable Code", The Computer Journal, Vol. 23, No. 1, pp.66-72.

Evans, D.J. [1984]: "New ParalleZ Algomithms for Partial Differential Equations", In 'Parallel Computing $83^{\prime}$, edited by Feilmeier, M., Joubert, J. and Schendel, U., Elsevier Science Publishers, B.V. (North-Holland), Int. Conf. Parallel Computing 83, 1984, pp. 3-56.

Evans, D.J. [1985a]: "Group Explicit Iterative Methods for Solving Large Linear Systems", Intern. J. Computer Math., Vol. 17, pp.81-108.

Evans, D.J. [1985b]: "Matrix Iterative Methods: An Introduction", Internal Report 222, Computer Stuđies Dept., Loughborough University.

Evans, D.J. and Yousif, N. [1985]: "Parallel Neighbour Sort with the Parallel 2-Way Merge Algowithm", Internal Report 224, Computer Studies Dept., Loughborough University.

Feilmeier, M. [1982]: "ParalleZ Numerical Algomithms", In Parallel Processing System, edited by Evans, D.J., Cambridge University Press, 1982, pp. 285-338.

Flynn, M.J. [1966]: "Very-High-Speed Computing System", Proceedings of the IEEE, December, 1966, Vo1. 54, pp. 1901-1909.

Flynn, et al. [1970]: Flynn, M.J., Podvin, A. and Shimizu: "A Multiple Instmuction Stream with Shared Resources", In Parallel Processer Systems. Technologies and Applications, edited by Hobbs, L.C., Washington D.C., Spartan Books, pp. 251-286.

Flynn, M.J. [1972]: "Some Computer Organizationsand Their Effectiveness", IEEE Transactions on Computers, Sept. 1972, pp. 948-960.

Gavril, F. [1975]: "Merging with Parallel Processors", CACM, Vol. 18, No. 10, October 1975.

Gill, S. [1958]: "Parallel Programming", Computer Journal, Vol. 1, pp. 2-10, 1958.

Gerald, C.F. [1978]: "Applied Numerical Analysis", Addison-Wesley Publishing Company, Reading, Mass., 2nd ed.

Gonzalez, M.J. and Ramamoorthy, C.V. [1970]: "Recognition and Representation of Parallel Processable Streams in Computer Programs", in Parallel Processing Systems, Technologies and Applications, edited by Hobbs, L.C., Spartan Books, pp.335-373.

Gosden. J.A. [1966]: "Explicit Parallel Processing Description and Control in Programs for Multi- and Uni-processor Computers", In

Proc. AFIPS 1966 Fall Joint Computer Conf., Spartan Books, New York, pp. 651-660.

Hageman, L.A. and Kellogg, R.B. [1968]: "Estimating the Optimal OverRelaxation Parameter", Math. of Comp., Vol. 22, pp. 60-68.

Handler, W. [1977]: "The Impact of Classification Schemes on Computer Architecture", Proc. 1977 Int. Conf. on Parallel Proc., pp.7-15.

Hansen, P.B. [1972]: "Structured Multiprogramming", CACM, Vol. 15, No. 7, pp. 574-578.

Hansen, P.B. [1975]: "The Programming Language Concurrent Pascal", IEEE Trans. on Software Engineering, Vol. 1, No. 2.

Hansen, P.B. [1977]: "The Architecture of Concurrent Programs", Prentice-Hall, Englewood Cliffs, N.Y., 1977.

Hayes, J.P. [1983]: "Computer Architecture and Organization", McGraw Hill International Book Company, 4th Printing, 1983.

Heller, D. [1978]: "A Survey of Parallel Algorithms in Numerical Linear Algebra", SIAM Review, Vol. 20, No. 4, pp. 740-777.

Hellerman, H. [1966]: "Parallel Processing of Algebraic Expressions", IEEE Trans. on Electronic Computers, Vol.ECl5, No. 1, Feb. 1966.

Hibbard, T.N. [1963]: "An Empirical Study of Minimal Storage Sorting", CACM, Vol. 6, No. 5, pp. 206-213.

Hinz, R.G. and Tate, D.P. [1972]: "Control Data STAR-100 Processor Design", Compcon. 72, IEEE Computer Society Conference Proc., pp.1-4, Sept. 1972.

Hirschberg, D.S. [1978]: "Fast Parallel Sorting Algorithms", CACM, Vol. 21, No. 8, pp. 657-661.

Hoare, C.A.R. [1972]: "Towards a Theory of Parallel Programing", In Operating Systems Techniques, edited by Hoare, C.A.R. and Perrott, R., Academic Press, New York, 1972.

Hoare, C.A.R. [1974]: "Monitors: An Operating System Structuring Concept", CACM, Vol. 17, No. 10, pp. 549-557.

Hockney, R.W. and Jesshope, C.R. [1981]: "Parallel Computers: Architecture, Programming and Algorithms", Adam Hilger Ltd., Bristol.

Horowitz, E. and Zorat, A. [1983]: "Divide-and-Conquer for ParalZeZ Processing", IEEE Trans. on Computers, Vol. C-32, No. 6, June 1983, pp. 582-585.

Hsiao, D.K. and Menon, M.J. [1980]: "Parallel Record-Sorting Methods for Hardware Realization Systems", Technical Report OSU-CISRC-TR-80-7, The Ohio State University, Columbus, Ohio, July 1980.

Hyafil, L. and Kung, H.T. [1974]: "Parallel Algorithms for Solving Triangular Linear Systems with Small Parallelism Parameter", Dept. of Computer Science, Carnegie Mellon University.

Hwang, K. and Briggs, F.A. [1984]: "Computer Architecture and Parallel Processing", McGraw-Hill Computer Science Series.

Jensen, et al. [1976]: Jensen, E.D., Thurber, K.J. and Schneider, G.M.: "A Review of Systematic Methods in Distributed Processor Interconnection", IEEE Internat. Conf. on Communications, 1976, IEEE, N.Y.

Jesshope, C.R. and Craigie, J. [1980]: "Another Matrix Algorithm for the DAP", DAP Newsletter, vol. 4, pp. 7-14.

Joubert, G.R. and Maeder, A.J. [1982]: "An MIMD Parallel Computer System", Computer Physics Communications, 26, pp. 253-257.

Karp, R.M. and Miller, R.E. [1966]: "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing", SIAM J. Appl. Math., Vol. 14, No. 6, Nov. 1966.

Keller, R.M. [1970]: "On Maximally Parallel Schemata", In Conf. Rec., 1970 IEEE 1lth Ann. Symp. Switching and Automata Theory, pp.32-50.

Knuth, D.E. [1973]: "The Art of Computer Programing: Vol. 3 Sorting and Searching", Addison-Wesley Publishing Company, Reađing, Massachusetts.

Kogge, P.M. [1974]: "Parallel Solutions of Recurrence Problems", IBM J.Res. Develop., 18, pp. 138-148, March 1974.

Kotov, V.E. [1976]: "Theory of Parallel Programming: Survey of Practical Aspects", in Advances in Information System Science, edited by Tou, J.T., Vol. 6, 1976, Plenum Press.

Kuck, D.J. and Maruyama, K. [1975]: "Time Bounds on the Parallel Evaluation of Arithmetic Expressions", SIAM J. Comput., 4, pp. 147-162.

Kuck, D.J. [1977]: "A Survey of Parallel Machine Organization and Programming", Computing Surveys, vol. 9, No. 1, March, 1977.

Kung, H.T. [1976]: "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors", in Algorithms and Complexity, New Directions and Recent Results, edited by Traub, J.F., Academic-Press, pp. 153-200.

Kung, H.T. [1979]: "Let's Design Algorithms for VLSI Systems", Proc. of Conf. on VLSI: Architecture Design Fabrication, CALTECH, pp.56-90.

Kung, H.T. [1980]: "The Stmucture of Parallel Algorithms", Advances in Computers, Vol. 19, pp. 65-112, Academic Press, New York.

Kung, H.T. [1982al: "Why SystoZic Arrays?", IEEE Computer, Vol. 15, No. 1, pp. 37-46.

Kung, H.T. [1982b]: "Notes on VLSI Computation", in Parallel Processing Systems, edited by Evans, D.J., Cambridge University Press, pp. 339-356.

Kung, S.Y. [1985]: "VLSI Array Processors", IEEE ASSP Magazine, pp. 4-22.

Lawrie, et al. [1975]: Lawrie, D.H., Layman, T., Baer, D. and Randal, J.M.: "GLYPNIR - A Programming Language for ILLIAC-IV", CACM, Vol. 18, No. 3, March 1975, pp. 157-164.

Lorin, H. [1975]: "Sorting and Sorting Systems", Addison-Wesley Pub. 1975.

Maruyama, K. [1973]: "The Parallel Evaluation of Matrix Expressions", IBM T.J. Watson Research Centre, York Town Heights, N.Y.

McGettrick, A.D. [1978]: "Algol 68 A First and Second Course", Cambridge Computer Science Text 8, Cambridge University Press.

Mead, C. and Conway, L. [1980]: "Introduction to VLSI Systems", Addison-Wesley Series in Computer Science, Reading, Massachusetts, 1980.

Miklosko, J. [1984]: "Synthesis of Parallel Numerical Algorithms", In Algorithms, Software and Hardware of Parallel Computers, eds. Miklosko, J. and Kotov, V.E., Springer-Verlag, pp. 13-43, 1984.

Miller, R.E. and Cocke, J. [1972]: "Configurable Computers: A New Class of General-Purpose Machine", Int.Symp. on Theoretical Programming, Novosibirsk 1972. In Lecture Notes in Computer Science, Vol. 5, Springer-Verlag, New York, 1974, pp. 86-298.

Miller, R.E. [1973]: "A Comparison of Some Theoretical Models of Parallel Computation", IEEE Trans. on Computers, Vol. C-22, No. 8 , August 1973, pp. 710-717.

Millstein, R.E. [1973]: "Control Structure in ILLIAC IV Fortran", CACM, Vol. 16, No. 10, 1973, pp. 622-627.

Miranker, W.L. [1971]: "A Survey of ParalleZism in Numerical Analysis", SIAM Review, Vol. 13, No. 4, 1971, pp. 524-547.

Muller, D.E. and Preparata, F.P. [1975]: "Bounds for Complexity of Networks for Sorting and Switching", JACM, Vo1. 22, No. 2, April 1975, pp. 195-201.

Muraoka, Y. and Kuck, D.J. [1973]: "On the Time Required for a Sequence of Matrix Products", CACM 16, No. 1, 1973, pp.22-26.

Nassimi, D. and Sahni, S. [1979]: "Bitonic Sort on a Mesh-Connected Parallel Computer", IEEE Trans. on Computers, Vol. C-27, No. 1, Jan. 1979, pp.2-7.

Opler, A. [1965]: "Procedure-Oriented Language Statements to Facilitate Parallel Processing", CACM, Vol. 8, No. 5, May 1965, pp. 306-307.

Papernov, A.A. and Stasevich, V. [1965]: "A Method of Information Sorting in Computer Memories", Problemy Peredachi Informatsii, Vol. 1, No. 3, pp. 81-98, 1965.

Parter, S.V. [1961]: "Multi-Zine Iteration Methods for Elliptic Difference Equations and Fundamental Erequencies", Numer.Math. Vol. 3, pp. 305-319.

Paul, G. and Wilson, M.W. [1975]: "The VECTRAN Language: An Experimental Language for Vector/Matrix Array Processing", IBM Palo Alto Scientific Center Report 6320-3334, August, 1975.

Peaceman, D.W. and Rachford, H.H. [1955]: "The Numerical Solution of Parabolic and Elliptic Differential Equations", J.Soc.Indust.Appl. Math., Vol. 3, pp. 38-41.

Perrott, R.H. [1979]: "A Language for Array and Vector Processors", ACM Trans. on Programming Languages and Systems, Vol. 1, No. 2, Oct. 1979, pp. 177-195.

Perrott, R.H. [1980]: "Language for Parallel Computers", In on the Construction of Programs, ed. by McKeag, R.M., and MacNaughten, A.M., Cambridge University Press, pp. 255-281.

Perrott, R.H. [1982]: "Languages for Vector and Parallel Processors", Computer Physics Communications, vol. 26, 1982, pp. 267-275.

Peterson, J.L. [1977]: "Petri Nets", Computing Surveys, Vol. 9, No. 3, Sept. 1977, pp. 223-252.

Petri, C.A. [1966]: "Kommunikation mit Automaten", Schriften des Rheinisch-Westfalischen Institutes fur Instrumentelle Mathematik an der Universitat Bonn, Heft 2, Bonn, w. Germany 1962; translation: C.F. Greene, Supplement 1 to Tech.Rep. RADC-TR-65-337, Vol. 1, Rome Air Development Center, Griffiss Air Force Base, N.y. 1965, 89pp.

Podsiadlo, D.A. and Jordan, H.F. [1981]: "Operating System Support for the Finite Element Machine", Proc. of CONPAR, Lecture Notes in Computer Science, No. 1ll, Springer Verlag.

Pratt, V.R. [1972]: "Shellsort and Sorting Networks", Ph.D. Thesis, Computer Science, Stanford University, 1972.

Preparata, F.P. [1978]: "New Parallel Sorting Schemes", IEEE-TC, Vol. C-27, No. 7, July 1978.

Ramamoorthy, C.v. and Li, H.F. [1977]: "Pipeline Architecture", Comp. Survey, Vol. 9, No. 1, March 1977, pp. 61-102.

Rumbaugh, J. [1977]: "A Data Flow Multiprocessor", IEEE Trans. on Computers, Vol. C-26, No. 2, pp.138-146.

Russell, R.M. [1978]: "The Cray-1 Computer System", CACM; Vol. 21, No. 1, Jan. 1978, pp. 63-72.

Sajjan, G.S. [1985]: "Computer: Design and Structure", Little, Brown and Company (Canada) Ltd., 1985.

Sameh, A.H. [1977]: "Numerical Parallel Algorithms - A Survey", In High Speed Computer and Algorithm Organization, edited by D.J. Kuck, D.H. Lawrie and A.H. Sameh, Academic Press, New York, pp. 207-228, 1977.

Sameh, A.H. and Brent, R.P. [1977]: "Solving Triangular Systems on a ParalleZ Computer", SIAM J.Numer.Anal. Vol. 14, No. 6, pp. 11011113, 1977.

Seitz, C.L. [1982]: "Ensemble Architecture for VLSI - A Survey and Taxonomy", In Proc. of MIT Conf. on Advanced Research in VLSI, ed. P. Penfield, Jr., Artech House, January, 1982, pp. 130-135.

Sharp, J.A. [1985]: "Data Flow Computing", Published by: Ellis Horwood Limited, 1985.

Shell, D.L. [1959]: "A High-Speed Sorting Procedure", CACM, Vol. 2, No. 7. 1959, pp. 30-33.

Shore, J.E. [1973]: "Second Thoughts of Parallel Processing", Computing and Electrical Engineering, Vol. 1, Pergamon Press, oxford, England, 1973, pp. 95-105.

Slutz, D.R. [1968]: "The Flow Graph Schemata Model of Parallel Computation", Ph.D. Dissertation, Elect.Enge.Dept., MIT, Cambridge Mass., Sept. 1968.

Slutz, D.R. [1970]: "Flow Graph Schemata", in Rec. Project MAC Conf. Concurrent Syst. and Parallel Computation. New York, Ass.Comput. Mach., 1970, pp. 129-141.

Smith, G.D. [1978]: "Numerical Solution of Partial Differential Equations: Finite Difference Methods", 1978, second Edition, Oxford University Press, England, 1978.

Snyder, L. [1982]: "Introduction to the Configurable Highly Parallel Computer", IEEE Computer, vol. 15, pp. 47-56.

Snyder, L. [1984]: "Supercomputers and VLSI: The Effect of Large-Scale Integration on Computer Architecture", in Advances in Computers, Vol. 23, edited by M.C. Yovits, 1984, Academic Press, Inc.

Squire, J.S. [1963]: "A Transaction Algorithm for All Multiprocessor Computer", Proc. l8th ACM National Conf. 1963.

Stone, H.S. [1967]: "One Pass Compilation of Arithmetic Expressions for a ParalleZ Processor", CACM, Vol. 10, No. 4, Apri1 1967, pp.220-223.

Stone, H.S. [1971]: "Parallel Processing with the Perfect Shuffle", IEEE Trans. on Computers, Vol. C-20, Feb., 1971, No. 2, pp. 153-161.

Stone, H.S. [1973]: "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations", J. of ACM, Vol. 20 , No. 1, January 1973, pp. 27-38.

Stone, H.S. [1973a]: "Problems of Parallel Computation", In Complexity of Sequential and Parallel Algorithms, edited by J.F. Traub, Academic Press, New York, 1973, pp. 1-16.

Tang, C.Y. and Lee, R.C. [1984]: "Optimal Speeding Up of Parallel Algorithms Based Upon the Divide-and-Conquer Strategy", Information Science, Vol. 32, pp. 173-186.
[Texas Instruments, I]: "Model 990 Computer, DX10 Operation System Release 3 Reference Manual, Concept and Facilities", Vol. I, Computer Studies Dept., Loughborough University.
[Texas Instruments, II \& IV]: "Texas Instmuments DX10 Operating System Reference Manual, Vol. II\& IV', Computer Studies Dept., Loughborough University.
[Texas Instruments, VI]: "Texas Instruments DX10 Operating System Reference Manual, Vol. VI", Computer Studies Dept., Loughborough University.

Thomas, C.D. and Kung, H.T. [1977]: Sorting on a Mesh-Connected Parallel Computer", CACM, VO1. 20, No. 2, 1977, pp. 263-271.

Thurber, K.J. [1976]: "Large Scale Computer Architecture: Parallel and Associative Processors", Hayden Publishing, Rochelle Park, N.J., 1976.

Thurber, K.J. [1979]: "Parallel Processor Architecture - Part I: General Purpose Systems", Computer Design, January, 1979.

Valiant, L.G. [1975]: "ParalleZism in Comparison Problems", sIAM J. of Computing, Vol. 3, No. 4, Sept. 1975, pp. 348-355.

Varga, R.S. [1962]: "Matrix Iterative Analysis", Prentice-Hall, Englewood Cliffs, N.J., 1962.

Vichnevetsky, R. [1981]: "Computer Methods for Partial Differential Equations", Vol. 1, Prentice-Hall, Inc., N.Y. Englewood Cliffs, N.J., U.S.A. 1981.

Watson, W.J. [1972]: "The TI ACS, A Highly Modular and Flexible SuperComputer Architecture", AFIPS Fall 1972, Press, Montvale, N.J., Vol. 41, Part 1, pp. 221-229.
williams, S.A. [1978]: "Approaches to Determination of Parallelism in Computer Programs", Ph.D. Thesis, Computer Studies Department, Loughborough University of Technology.

Wyllie, J.C. [1979]: "The Complexity of Parallel Computation", Ph.D. Thesis, Computer Science Dept., Cornell University, U.S.A., 1979.

Yau, S.S. and Fung, H.S. [1977]: "Associative Processor Architecture A Survey", Computing Surveys, vol. 9, No. 1, March 1977, pp. 3-27.

Young, D.M. [1954]: "Iterative Methods for Solving Partial Differential Equations of Elliptic Type", Trans.Amer.Math.Soc., Vol. 76, pp. 92-111.

Young, D.M. [1971]: "Iterative Solution of Large Linear Systems", Academic Press, New York, 1971.

Yousif, N.Y. [1983]: "Parallel Algoxithms for Asynchronous Multiprocessors", Ph.D. Thesis, Computer Studies Dept., Loughborough University of Technology, 1983.

Yousif, N.Y. and Evans, D.J. [1985a]: "The ParalleZ Odd-Even Merge Algorithm", Internal Report 223, Computer Studies Dept., Ioughborough University, 1985.

Yousif, N.Y. and Evans, D.J. [1985b]: "Merging by the Parallel Binary Search Algorithm", Internal Report 225, Computer Studies Dept., Loughborough University, 1985.

Yousif, W.S. [1984]: "New Block Iterative Methods for the Numerical Solution of Boundary Value Problems", Ph.D. Thesis, Dept. of Computer Studies, Loughborough University of Technology, 1984.


[^0]:    resource allocation on the basis of some advance information about resource usage so that the deadlock is avoided. With deadlock detection and recovery, the scheduler gives resources to the process as soon as they become available and, when deadlock is detected, the scheduler pre-empts some resources in order to recover the system from the deadlock situation.

[^1]:    Among those researchers who have studied the implementation of sorting algorithms on the SIMD design are Baudet and Stevenson [1978] who presented a generalized odd-even transposition. Nassimi and Sahni [1979] also presented a $O(n)$ algorithm to sort $n^{2}$ elements on an $n \times n$ mesh-connected parallel computer and Thompson and Kung [1977] developed an algorithm for sorting $n^{2}$ elements on an $n \times n$ mesh-connected processor array that requires $O(n)$ routing and comparison steps.

[^2]:    *See Chapter 4 for definition.

[^3]:    *See Chapter 4 for the definition.

[^4]:    *See Chapter 4 for definition.

[^5]:    speed-up ratios on parallel computers depending on whether the implementation is done synchronously or asynchronously. In both cases, the overhead measurements incurred when more than one processor is cooperating should be borne in mind. 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, such as processors, shared data or a memory block. Finally, the experimental results that are obtained from the different implementations of the 9-point and 4-point block methods on the NEPTUNE system are presented and analysed.

[^6]:    Evans and Biggins [1982] show that,

[^7]:    $\left.\begin{array}{l} \\ \\ \\ \\ \end{array}\right]$

