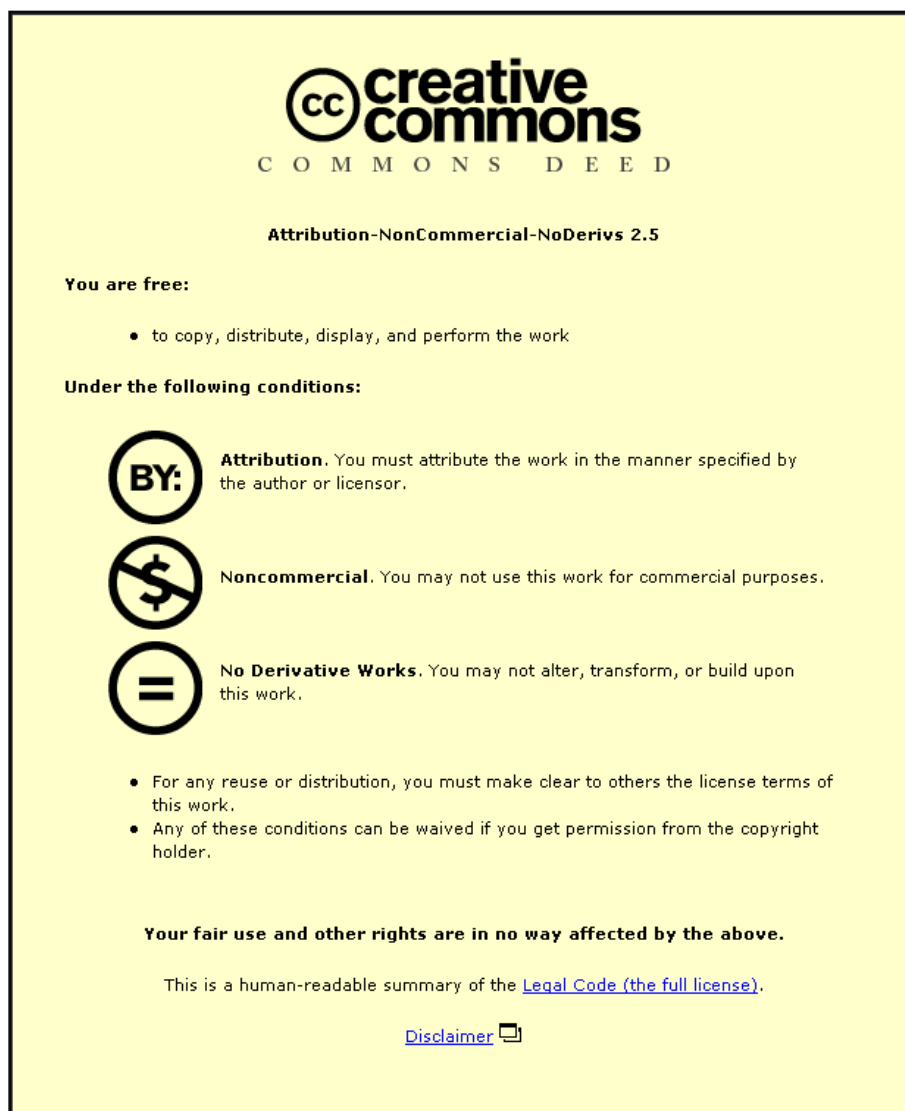


This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (<https://dspace.lboro.ac.uk/>) under the following Creative Commons Licence conditions.



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

BWDSC NO :- DX 76089.

LOUGHBOROUGH
UNIVERSITY OF TECHNOLOGY
LIBRARY

AUTHOR/FILING TITLE

GHANEMI, S

ACCESSION/COPY NO.

014522/02

VOL. NO.

CLASS MARK

4 JUL 1988	LOAN COPY
- 1 JUL 1988	- 6 JUL 1990
30 JUN 1989	- 5 JUL 1991
30 JUN 1989	- 2 JUL 1993
30 JUN 1989	

001 4522 02



NON-NUMERICAL PARALLEL ALGORITHMS FOR ASYNCHRONOUS PARALLEL COMPUTER SYSTEMS

by

SALIM GHANEMI, B.Eng, MSc

A Doctoral Thesis
Submitted in partial fulfilment of the
requirements for the Award of
Doctor of Philosophy
of the
Loughborough University of Technology

Sept. 1987

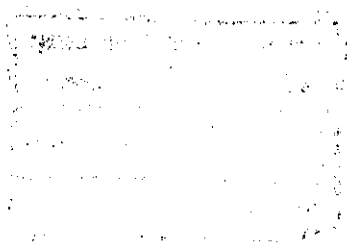
Supervisor: Professor David Jones Evans, PhD, DSc
Department of Computer Studies

Loughborough University	
of Technology Library	
Date	NW 87
Class	
Acc. No.	014522/02

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 University or any other institution for a higher degree.

Salim Ghanemi



ACKNOWLEDGEMENTS

The author wishes to express his sincere thanks to Professor D J Evans for his guidance, suggestions and advice during the past three years of study and preparation of this thesis.

The author also acknowledges the Algerian Ministry of Higher Education (MES) for their financial support.

Thanks also to my parents for giving me the incentive to start and complete my research.

Finally, thanks to Mr R P Stallard for his help in maintaining the parallel systems operational.

ABSTRACT

The work in this thesis covers mainly the design and analysis of many important Non-Numerical Parallel Algorithms that run on MIMD type parallel computer systems (PCS), in particular the NEPTUNE and the SEQUENT BALANCE 8000 PCS available at Loughborough University of Technology.

Initially, different types of existing PCS including data-flow computers and VLSI technology are described from both the hardware and software points of view. Some basic ideas of efficiently programming such computers are also presented. Also the main characteristics of both available systems, the Neptune and the Balance 8000, are outlined with the principles of synchronisation, the resource demands and the overheads of the parallel control structures. Such information is frequently measured in the performance analysis of the algorithms presented in this thesis in order to exploit the potentiality of the available systems and PCS in general.

In this study, some computer search problems with or without broadcasting are investigated. For example, the search of ordered and unordered files by key comparison are studied where the number of processes are greater or equal to the number of available processors. Binary and jump searching algorithms are also studied. The design, implementation and comparison of the parallel pattern and string matching algorithms are also included. Parallel sorting algorithms are studied and compared with two newly developed parallel partitioned sorting algorithms which show a higher performance rating than the parallel Quicksort algorithm.

The problem of pattern matching has been selected for the design of soft-systolic algorithms. Several versions of the pattern matcher are implemented using new design ideas which are projected to be materialised in the near future.

DEDICATED TO

*My Wife and Love Rabia,
for her constant support during
the course of this work*

CONTENTS

	<u>Page No</u>
Declaration	i
Acknowledgements	ii
Abstract	iii
 CHAPTER 1: MOTIVATIONS AND EXPLOITATIONS OF VARIOUS FORMS OF PARALLELISM	 1
1.1 Introduction	1
1.2 Main Motivations	6
1.3 Design Classifications	8
1.3.1 Flynn's high-speed parallel computers classification	 8
1.3.2 Shore's classification	16
1.3.3 Other classification approaches	19
1.4 Pipelined Computers	21
1.4.1 The pipelined principle and perfor- mance characteristics	 24
1.4.2 Vector processing	33
1.4.3 Implemented pipelined computers	35
1.5 Data-Flow Machines	45
 CHAPTER 2: CURRENT MULTIPLE PROCESSOR SYSTEM ARCHITEC- TURES	 51
2.1 Introduction	51
2.2 The Genealogy of the SIMD Computers	52
2.2.1 The utilisation and applications of the SIMD system	 53
2.2.2 Associative processors	56
2.2.2.1 Associative memory organisa- tion	 58
2.2.2.2 The taxonomy of associative processors	 61

	<u>Page No</u>
i) Fully-parallel	61
ii) Bit-serial	64
iii) Word-serial	67
iv) Block-oriented	69
2.2.3 Array processors	71
2.2.3.1 Interleaved parallel memories	74
2.2.3.2 The interconnection networks	76
2.2.3.3 Implemented array processor computers	79
2.3 MIMD Multiprocessor Computers	86
2.3.1 MIMD hardware organisation	87
2.3.2 Operating system organisation	94
2.3.3 Implemented MIMD multiprocessor systems	97
2.3.3.1 The INTERDATA DUAL processor	98
2.3.3.2 The NEPTUNE parallel computer	101
2.3.3.3 The SEQUENT BALANCE 8000 system	107
 CHAPTER 3: PROGRAMMING TOOLS AND PERFORMANCE ANALYSIS OF PARALLEL ALGORITHMS	113
3.1 Parallel Detection	113
3.1.1 Implicit parallelism	116
3.1.2 Explicit parallelism	120
3.2 Parallel Programming Supports of the Loughborough MIMD Systems	129
3.2.1 The user-interface to the Neptune system	141
3.2.2 Parallel control scheme in the Neptune system	147
3.3 Performance Characteristics Measurements of Parallel Algorithms	155
3.4 General Parallel Program Structuring Concepts	166

	<u>Page No</u>
CHAPTER 4: PARALLEL SEARCHING ALGORITHMS	188
4.1 Introduction	188
4.2 An MIMD Implementation of the Sequential Search Algorithm	189
4.2.1 Parallel sequential search without broadcasting	198
4.2.2 Parallel sequential search with broadcasting	201
i) Analysis of Version 1.0	202
ii) Analysis of Version 2.0	210
4.3 A Parallel Implementation of the Binary Search	213
4.3.1 Parallel binary search versions 1.0 and 2.0	218
4.3.2 Parallel binary search version 3.0	224
4.4 Parallel Jump Search Algorithms	228
4.4.1 Implementation and analysis	231
4.4.2 Experimental results	234
4.5 Conclusion	243
CHAPTER 5: PARALLEL STRING MATCHING ALGORITHMS	245
5.1 Introduction	245
5.2 History	247
5.3 Design, Analysis and Implementation	250
5.3.1 Parallel Brute Force algorithm	255
5.3.2 Parallel Knuth-Morris-Pratt algorithm (PKMP)	258
5.3.3 Parallel Karp-Rabin algorithm (PKR)	260
5.3.4 Parallel Boyer-Moore algorithm (PEM)	261
5.4 Conclusions	265
CHAPTER 6: PARALLEL SORTING ALGORITHMS	267
6.1 Introduction	267
6.2 Performance Analysis when $M=P$	271

6.2.1	Description of the sequential QUICKSORT algorithm	271
6.2.2	Parallel QUICKSORT algorithm (PQ) . .	274
6.2.3	Parallel QUICKSORT-MERGE (PQM)	283
6.2.4	Parallel Partitioned Sorting algorithms (PPS)	287
6.3	Performance Analysis when $M > P$	293
6.3.1	Parallel QUICKSORT algorithm (PAQ) . .	294
6.3.2	Parallel QUICKSORT-MERGE algorithm (PQM)	300
6.3.3	Parallel Partitioned Sorting algorithms (PPS)	306
6.4	Conclusions	308
CHAPTER 7:	A VLSI SOFT-SYSTOLIC IMPLEMENTATION OF A STRING MATCHER AND ITS VARIANTS	310
7.1	Introduction to the VLSI Technology Paradigm	310
7.2	Fundamental Architecture Concepts in Designing Special-Purpose VLSI Computing Structures	314
7.2.1	Systolic arrays	317
7.2.2	Wavefront arrays	322
7.3	VLSI-Oriented Architecture	325
7.3.1	The WARP architecture	325
7.3.2	The CHIP architecture	328
7.3.3	The INMOS transputer and OCCAM	331
7.3.4	Simulation of systolic arrays	334
7.4	Systolic Algorithms, Constraints and Classifications	339
7.5	Systolisation of the Pattern Matching Problem and its Variants	342
7.5.1	Hard-systolic designs	344
7.5.2	Soft-systolic designs	348

	<u>Page No</u>
i) Soft-systolic designs with broad-casting	349
ii) Soft-systolic designs when results are fanned-in	354
7.5.3 Conclusion	356
CHAPTER 8: CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK	359
REFERENCES	368
APPENDICES:	
Appendix A: Selected Parallel Programs	387
Appendix B: Summary of the OCCAM Language	435
Appendix C: Loughborough OCCAM Compiler Version 5.0 Documentation	444
Appendix D: Selected Systolic Programs	453

Chapter 1

MOTIVATIONS AND EXPLOITATIONS OF VARIOUS FORMS OF PARALLELISM

1.1 INTRODUCTION

Of all the previously witnessed scientific and technological revolutions that have greatly affected our lives in all its aspects, the **computer** or the information revolution, has had the most tremendous impact on both technology and our society. This fast developing revolution which has just recently started to migrate towards a new era, the knowledge revolution, by giving birth to the Fifth Generation of Super Computers (FGSC) has in fact changed our lifestyles, our educational programmes and most of all many of our professional careers. Similarly to the engine, a key element in the industrial revolution, the computer's innovation was motivated by the need to develop some device that could take over the repetitive mental work from the human mind. Thus, if the engine could be considered as the "**muscle**" then the microprocessor is, by all means, the "**brain**" for this era.

Amongst the huge numbers of computer applications which range from the simple personal computer games to the weather forecasting calculation and satellite transmission programs, there are many that require the use of large amounts of computational time. In an attempt to meet the challenging problem of providing fast and economical computation, the large-scale parallel computers were developed. In fact, until recently computational speed was derived only from the development of faster electronic devices.

The earliest computers built in the late 1950s used relatively slow devices such as **vacuum** tubes and their central memories were **magnetic** drums. As electronic technology advanced the demand for faster components was appreciated and therefore these were replaced by **transistors** and **magnetic** cores. In the late 1960s, Integrated

Circuits (ICs) were used in computer design and were followed by Large Scale Integrated (LSI) techniques. the Very Large-Scale Integrated circuits (VLSI), developed five years ago, are currently being used in the design of very high speed special and general purpose computer systems. The topic of VLSI systems is covered in Chapter 7 by outlining the main design methodologies. Examples of the design of a pattern matcher system are also included.

Until five years ago, the current state of electronic technology was such that all factors affecting computational speed were almost minimised and any further computational speed increase could only be achieved through both increased switching speed and increased circuit density. Due to the basic physical laws, the intended breakthrough seemed unlikely to be achieved mainly because we are fast approaching the limits of optical resolution. Hence, even if switching times are almost instantaneous, distances between any two points may not be small enough to minimise the propagation delays and thus improve computational speed. Therefore, the achievement of even faster computers is conditioned by the use of new approaches that do not depend on breakthroughs in device technology but rather on imaginative applications of the skills of computer architecture.

Obviously, one approach to increasing speed is through parallelism. The ideal objective is to create a system containing p processors, connected in some cooperating fashion, so that it is p times faster than a computer with a single processor. These parallel computer systems or multiprocessors as they are commonly known, not only increase the potential processing speed, but they also increase the overall throughput, flexibility, reliability and provide for the tolerance of processor failures. Unfortunately, the overheads associated with controlling and coordinating the effort of the p

processors often prevents the ideal speed and reliability improvements from occurring. Many of these overheads will be clearly described later in the thesis.

Parallelism, the notion of the parallel way of thinking was conceived long before the emergence of truly parallel computers. It is thought that the earliest reference to parallelism is in L F Manebrea's publication¹, entitled "Sketch of the analytical engine" invented by C Babbage. There, reporting on the utility of the conceived machine, he wrote: "Likewise, when a long series of identical computations is to be performed, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the process".

As we are moving towards fifth-generation-type systems, it is now possible to define the first four generations of computers which have been characterised in various ways but most commonly in terms of the technology of their hardware.

The first generation of computers (e.g. machines such as EDSAC² and Colossos³ were built out of thermionic valves with gate delay times of approximately 1 μ s (1 microsec). Systems were cumbersome,

-
1. Following Babbage's lecture in Turin, describing his "difference engine" a young Italian engineer wrote a detailed account of the machine in French (published in October 1842). Ada, Lady Lovelace translated the paper into English.
 2. In 1947, M Wilkes et al, from Cambridge, began the construction of the electronic Delay Storage Automatic Computer (EDSAC) and in May 1949 the system was operational.
 3. In 1943, Colossos, the first electronic computer, went into operation in Britain to decipher the messages produced by Enigma, a German code generator.

unreliable and needed ancillary cooling equipment to deal with the heat generated. Various software innovations (e.g. the operating system in EDSAC) were introduced during this phase.

Following the pioneering work (late 1940s) of W Shockley, J Bardeen and W Brattain at the Bell Laboratories in the US, the use of the germanium transistor, around 1960 with propagation delay times of approximately $.3 \mu\text{s}$, gave rise to the second generation of computers, such as the IBM 1401 and NCR 304.

The third generation of computers (e.g. IBM S/360 and ICL 1900) which were introduced around 1965, were built out of Integrated circuits (ICs): transistors, resistors, capacitors and diodes were fabricated in a $10 \mu\text{m}$ thick surface layer on a silicon wafer. The components were then connected by a metal layer evaporated onto the silicon, subsequent etching producing the required interconnections. These systems featured a propagation delay of 10 ns (nano-secs), and later, around 1975, of slightly less than 1 ns. By now, high-level programming languages (e.g. COBOL and Fortran) and sophisticated operating systems (e.g. IBM OS and ICL George 3) were well established.

The fourth-generation computers (e.g. IBM 3081 and Fujitsu M380) are characterised by enhanced levels of circuit integration through VLSI techniques. Various software and architectural innovations have been introduced, as well as new scientific terms came into use during this phase, such as on-line, real-time, multiprogramming, multiprocessing and asynchronous programming etc.

All these various new conceived multiple processor architectures whose aim is to increase the processing rate of a computer can be

characterised in four different categories: associative, parallel, pipelined and multiprocessors.

Hockney and Jesshope [Hockney 1981] summarised the principal ways of introducing parallelism at the hardware level of the various computer architectures as:

1. The application of pipelining - assembly lines - techniques in order to improve the performance of the arithmetic or control units. A process is decomposed into a certain number of elementary subprocesses each of which being capable of executing on dedicated autonomous units;
2. The provision of several independent units, operating in parallel, to perform some basic fundamental functions such as logic, addition or multiplications;
3. The provision of an array of processing elements performing simultaneously the same instruction on a set of different data where the data is stored in the processing element private memories;
4. The provision of several independent processors, working in a cooperative manner towards the solution of a single task by communicating via a shared or common memory, each one of them being a complete computer, obeying its own stored instructions.

The following sections will cover a wide selection of the principal significant parallel computer architectures, which differ sufficiently from each other, the pipeline, SIMD, MIMD, data-flow and VLSI systems, to illustrate alternative hardware and software

approaches. Specifically for the multiprocessor class, the Neptune and Balance 8000, at Loughborough University of Technology, are described in more detail, due to the fact that they were used extensively during the development of this present research.

1.2 MAIN MOTIVATIONS

During the last decade, the multiple processor approach has tailored a set of long sought after motivating goals in order to satisfactorily meet many of the challenging system design requirements. In reviewing some aspects of the parallel processing systems, one finds that while the hardware is improving at a fast rate, the software tools to take advantage of the provided benefits are only slowly forthcoming; a fact that affects the design motivations mentioned below.

Since the early developed multiple processing systems, the system characteristics that have motivated the continued development in this field have not changed much. The most significant of these are increased throughput, improved flexibility and reliability. Since None of these goals is numerically specified (i.e. they are all qualitative goals), it is not surprising that the design of the future "supercomputers" will also be motivated by the same objectives as today's parallel computers. However, the improvements of some or all of these specifications must ultimately result in an improved overall system performance, usually measured on the basis of cost effectiveness.

The system throughput can be used to mean several different characteristics such as the potential number of bits processed per time-unit, the number of memory transfers per time unit or the

maximal number of programs that can be handled at the same time. However, it is usually used nowadays to describe the low-turnaround of a program in a multiprocessing environment. The multiple processor approach is a cost-effective solution to the achievement of most of these goals. The use of several cooperating processing units can considerably increase the system throughput which could not be matched by a uniprocessor system with enhanced logic circuitry.

Literally, **flexibility** means the ease in changing the system configuration to suit new conditions and the use of more than one processor has greatly increased the system potential flexibility since it offers the ability to expand the memory space, the number of processing units and even the software facilities in order to meet the new demands. This flexibility may also be used to justify the increased **reliability** of the system.

Broadly speaking, the reliability is related to two different system aspects required by different applications. The first one is the **system availability** which is defined by the requirement that the system should remain available even in the case of a malfunctioning unit. An example of this is the computer controlled telephone switching board. The **system integrity** is the second one and it is defined by the requirement that the information contained within should be "protected" against any defection or corruption (e.g. in a banking system).

Concluding, since all the system characteristics that have motivated the development of the parallel processor computers are not described quantitatively, any new major system concept has been claimed by its proponents as the ultimate solution to achieving

these motivating goals. In fact, the same motives were behind the follow-up to the parallel processing systems, the VLSI architectures.

1.3 DESIGN CLASSIFICATIONS

As a result of the introduction of various forms of parallelism which has proved to be an effective approach for increasing computational speed, several competitive computer architectures were constructed but there was little evidence as to which design was superior, nor was there sufficient knowledge on which to make a careful evaluation. Researchers helped the study of high-speed parallel computers by attempting to classify all the proposed computer architectures, or at least those which have been already well established. A brief presentation of the concepts of the architectural taxonomy given by different researchers, especially by the two pioneers, Flynn [Flynn, 1966] and Shore [Shore, 1973], follows below. Although these classifications are not strict and complete since several classes could include the same computer (e.g. ICL DAP fits equally well into several different classified groups) or a computer could belong to any of them (e.g. pipelined computers), they have been widely mentioned and their corresponding terminology has greatly contributed to the formation of the computer science terminology.

1.3.1 FLYNN'S HIGH-SPEED PARALLEL COMPUTERS CLASSIFICATION

Based on the dependent relation between instructions that are propagated by the computer and the data being processed, Flynn explored theoretically some of the organisational possibilities for large scientific computing machinery before attempting to classify

them into four broad classes. We shall briefly review his theoretical concepts leading to the actual grouping of the high-speed parallel computers.

For convenience, he defined the instruction stream as a sequence of instructions to be processed by the computer and the data stream as a set of operands, including input and partial or temporary results. Also two additional useful concepts were adopted, bandwidth and latency. By bandwidth he expressed the time-rate of occurrences, and latency is used to express the total time between execution of response of a computing process on a particular data unit.

Particularly for the former notion, computational or execution bandwidth is the number of instructions processed per second and storage bandwidth is the retrieval rate of the data and instruction from the store (i.e. memory words per second).

By using the two former definitions, Flynn categorized the almost theoretically defined computer organisations depending on the multiplicity of the hardware provided to service the Instruction and Data streams. The word "multiplicity", which was intentionally used to avoid the ubiquitous and ambiguous term "parallelism", refers to the maximum number of simultaneous instructions or data in the same phase of execution at the most constrained component of the organisation.

Flynn observed that as a consequence of the above definitions four classes emerged naturally, being characterized from the multiplicity or not of the Instruction and Data Streams:

- i) Single Instruction stream - Single Data stream (SISD)

- ii) Single Instruction stream - Multiple Data stream (SIMD)
- iii) Multiple Instruction stream - Single Data stream (MISD)
- iv) Multiple Instruction stream - Multiple Data stream (MIMD)

The SISD computer [e.g. most of the general purpose machines such as IBM STRETCH, DEC PDP-11 (serial or unpipelined) and CDC 6600 series, IBM 360/90 series (pipelined)], is nothing more than the ordinary serial computer (the Von-Neumann type computer). Even though, the CDC 6600 and IBM 360/90 series achieve their power by overlapping various sequential decision processes which make up the execution of the instruction (Confluent SISD), there still remains an essential constraint of this type of organisation, namely the decoding of one instruction per unit time. In Figures 1.1 and 1.2 we see a SISD organisation, and the concurrency and instruction processing respectively.

The SIMD type structure, proposed by Unger [Unger 1958], Slotnick [Slotnick 1962] is created by replicating the data stream on which the single instruction stream acts simultaneously thus theoretically increasing the throughput by a factor almost equal to the number of data streams. Several factors, such as data conflict and data communication problems tend to degrade the expected performance. SOLOMON and ILLIAC IV are two examples of such a computer.

The third, MISD type class of parallel computers, the organisation of which is outlined in Figure 1.3, is by all means the least realistic one compared to the others since no examples of any well established organisation have yet been proposed. In this class, a forwarding procedure of data flowing through the Execution Units was forced. Thus, the data stream presented to Execution Unit 2 is the resultant of Execution Unit 1 operating its instruction on the

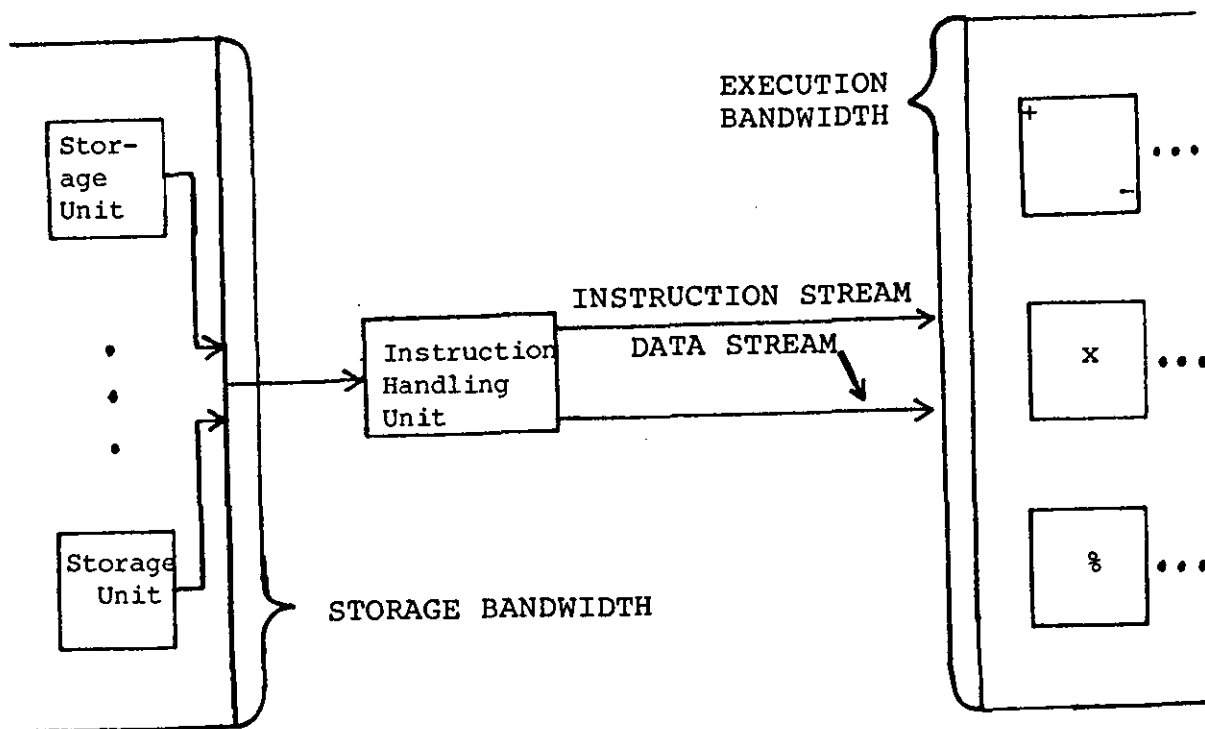


FIGURE 1.1: FLYNN'S SISD COMPUTER ORGANISATION

source data stream. The instruction performed on any Execution Unit can be one of the three following types: **fixed**, **semi-fixed** or **variable**. It may be **flexible** such that the interconnection of units must be flexible, **semi-fixed** such that the function of any unit is fixed for one pass of the data or **variable** meaning that the execution of a stream of instructions can take place at any point on the single data stream. Consequently this arrangement suggests that only the first processing component faces the source data stream whereas the remaining units are processing derivations of the data from previous components.

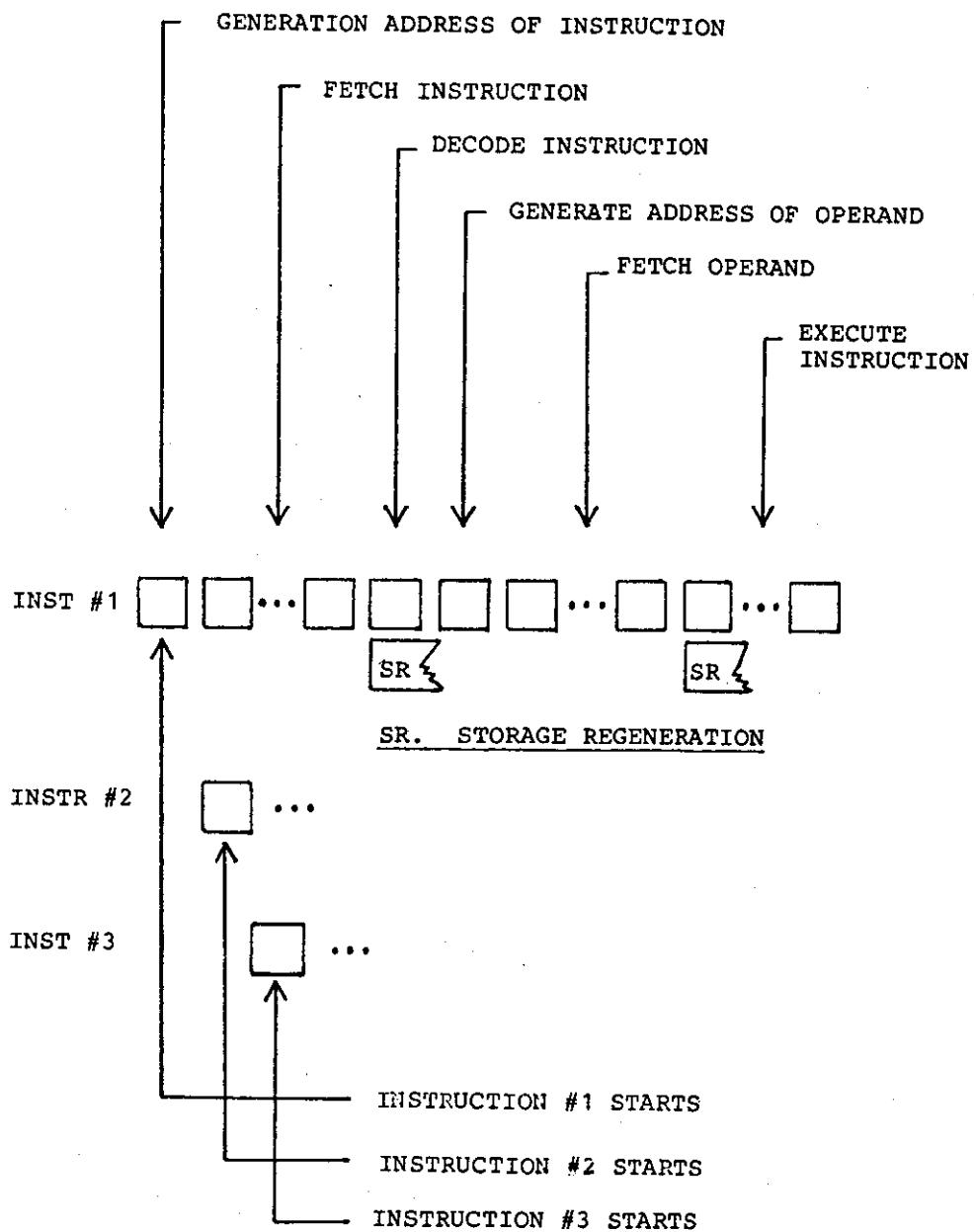


FIGURE 1.2: CONCURRENCY AND INSTRUCTION PROCESSING

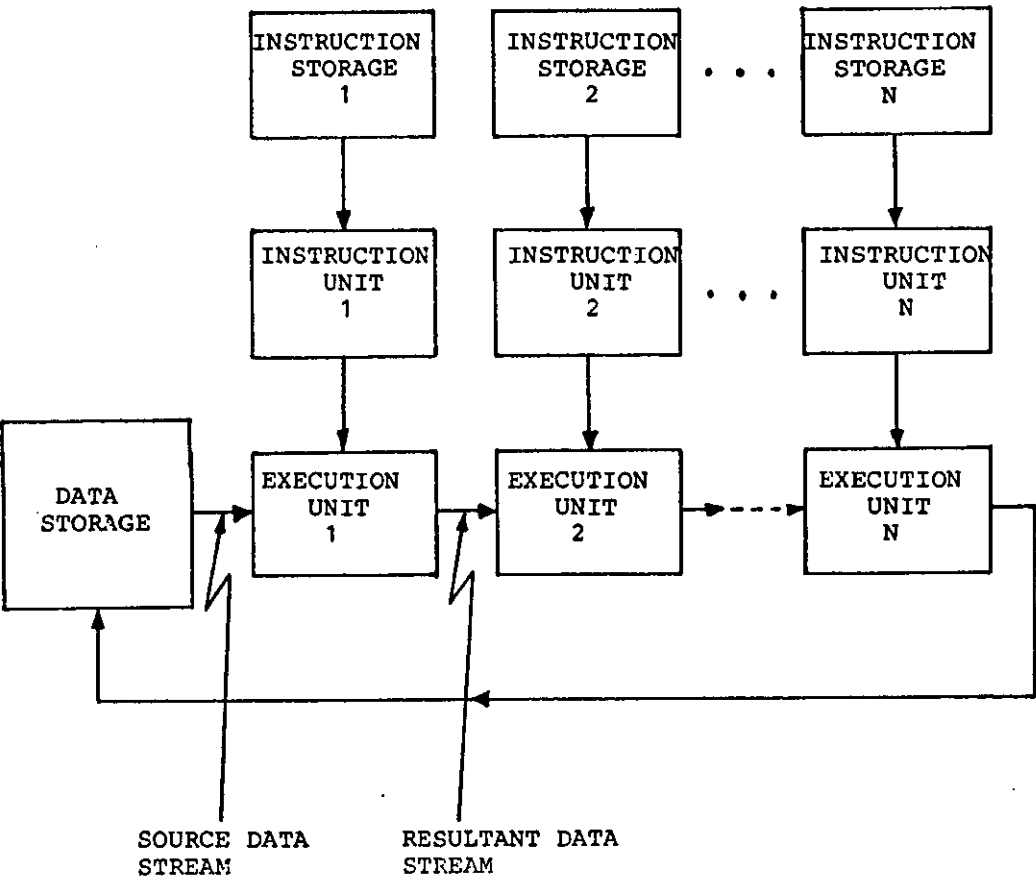


FIGURE 1.3: A MIMD ORGANISATION

By combining parallelism in both the instruction and data streams a MIMD type of structure is thus obtained. This computer possesses N independent executing units (processors), each of which is a complete computer on its own (has arithmetic and logic capabilities and local data storage), with processors connected together to provide means for cooperation during a computation phase.

Most serial main frames could be classified as MIMD computers since they include many data channels, such as Direct Memory Access (DMA) which are, in a sense, independent processors. Thus a computer with one or two data channels is indeed a MIMD parallel computer, but the MIMD is commonly accepted to refer to large computers with possibly several identical processors such as Ccmp [Wulf 1972], CM* [Swan 1977]. Of particular interest, the NEPTUNE and BALANCE 8000 parallel computer systems are examples of this class.

Resuming, Flynn classified computer systems into four broad classes (see Figure 1.4) depending on the multiplicity or not of the instruction stream and data stream. Due to the fact that the actual architectural details of the machines were not taken into account, his taxonomy was somehow obscure since one finds that there is no apparent distinctive differences between classes (MIMD class exempted). Consequently, pipelined and array processor computers are considered similar, although they are two completely different architectures.

Also, the meaning of the data streams, as used by Flynn, has caused many ambiguities due to the fact it does not make a distinctive difference between a single stream of vectorised data and a multiple scalar stream.

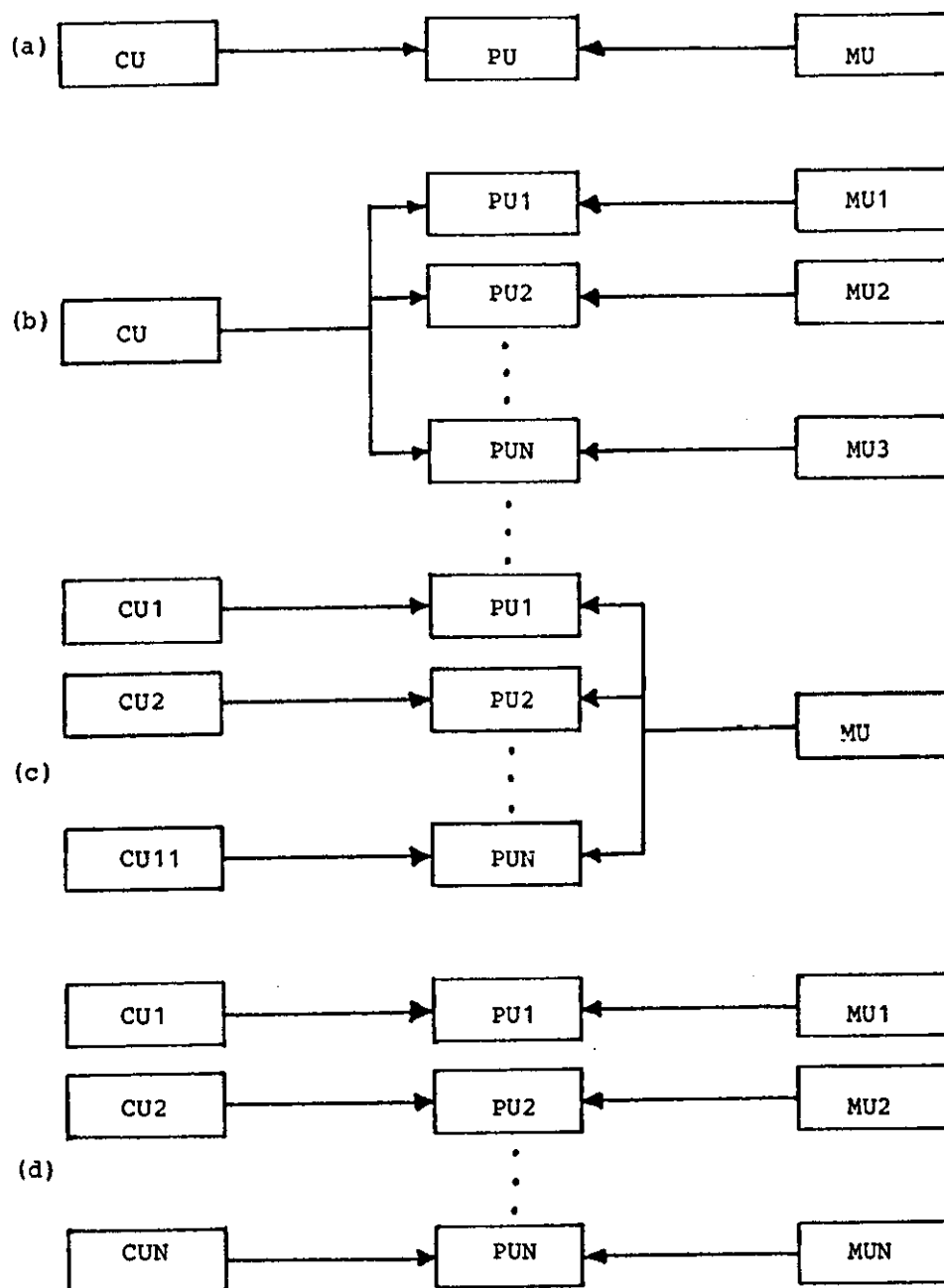


FIGURE 1.4: FLYNN'S COMPUTER ORGANISATION CLASSES
 (a) SISD; (b) SIMD; (c) MISD, and (d) MIMD
 where CU, PU and MU refer to Control, Processing
 and Memory Units respectively

Consequently, in the following sections, the SIMD and pipelined computers are considered to be two distinct classes along with the multiprocessor category.

1.3.2 SHORE'S CLASSIFICATION

Classification of parallel computer systems based on their constituent hardware components was observed by Shore [Shore 1973]. Accordingly, all current existing computer architectures were categorised into six different classes which are schematically shown in Figure 1.5.

The first machine (I), [e.g. CDC 7600, a pipelined scalar computer, CRAY 1, a pipelined vector computer] which is the conventional serial Von Neumann-type organisation, consists of an Instruction Memory (IM), a single Control Unit (CU), a Processing Unit (PU), and a Data Memory (DM). The main source of power increase comes from the processing unit which may consist of several functional units, pipelined or not and all bits of a single word are read in order to be processed simultaneously (Horizontal PU).

A second alternative machine (II) is obtained from the first one by simply changing the way data is read from the Data Memory. Instead of reading all bits of a single word as (I) does, machine (II) reads a bit from every word in the memory i.e. bit serially, but word processing is parallel. In other words, if the memory area is considered as a two dimensional array of bits, with each word occupying an individual row, then machine (I) reads horizontal slices whereas machine (II) reads vertical slices.

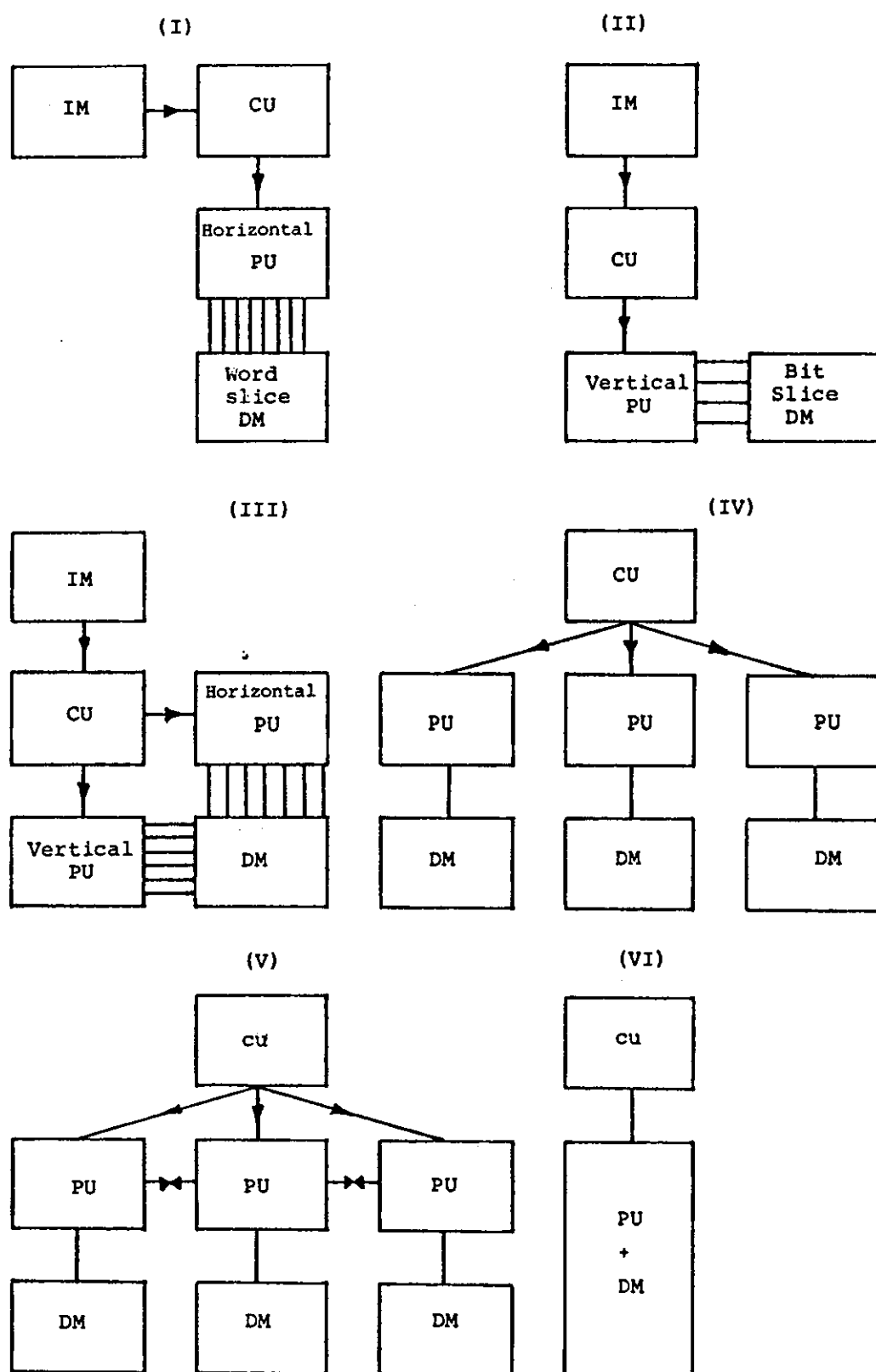


FIGURE 1.5: THE CONFIGURATION OF THE SIX MACHINE CLASSES

A combination of the two above machines yields machine III. This means that machine (III) has two processing units, a horizontal and a vertical one and is capable of processing data in either of the two directions. The ICL DAP could have been a favourable candidate for this class if only it had separate processing units to offer this capability. An example of this organisation is the Sanders Associates OMEN 60 series of computers [Higbie 1972].

Machine (IV) consists of a single control unit and many independent processing elements, each of which has a processing unit and a data memory. Communication between these components is restricted to take place only through the control unit. A good example of this machine is the PEPE system.

If, however, additional limited communication is allowed to take place among the processor elements in a nearest-neighbouring fashion, then machine V is conceived. Thus, communication paths between the linearly connected processors offer for any processor in the array the possibility to access data from its immediate neighbour memories, as well as its own. An example of this machine type is the ILLIAC IV, which provides a short cut communication every eight processing elements.

The Logic-In-Memory Array (LIMA) is Shore's last class of computer organisation. The main difference in machine (VI) and the previous one is that the processing unit and the data memory are no longer two individual hardware components, but instead they are constructed on the same IC board. Examples range from simple associative memories to complex associative processors.

It is observed that, generally speaking, Shore's classification, compared with Flynn's, does not offer anything new, but only a sub-categorisation of the obscure SIMD class given by Flynn, except for machine (I) which is an SISD-type computer. Again, as with Flynn's categorisation, pipelined computers do not belong to a well specified class, that represents their hardware characteristics, but on the contrary they are mixed up with unpipelined scalar computers.

1.3.3 OTHER CLASSIFICATION APPROACHES

This paragraph gives a brief note on some other classification approaches of less significant importance compared to the former two and which are based mainly on the concept of parallelism.

One of the taxonomies, based on the amount of parallelism involved in the control unit, data streams and instruction units was suggested by Hobbs et al [Hobbs 1970] in 1970. They distinguished parallel computers into multiprocessors, associative processors, array processors and functional processors.

Another classification, due to Murtha and Beadles [Murtha 1964] was based upon the parallelism properties. An attempt to underline the main significant differences between the multiprocessors and Highly Parallel organisations was appreciated. Three main classes for parallel processor systems were identified and they are general-purpose network computers, special-purpose network computers characterised by global parallelism and finally non-global, semi-independent network computers with local parallelism.

Furthermore, all these classes, but the last one, were further subcategorised into two subclasses each. Whereas the first class,

the general-purpose one, was subdivided into the general-purpose network computers subclass with centralized common control and the general-purpose network computers subclass, with many identical processors, each of which being capable of, independently from the others, executing instructions from its own local storage, the second class identified the Pattern processors and associative processors subclasses.

Using a structural chemistry-like notation¹, based on a shorthand indicating the number of instructions, execution and memory units, and the way they are interconnected and controlled, Hockney and Jesshope [Hockney 1981] formulated a taxonomy scheme for both serial and parallel computers. The main subdivisions are shown in Figures 1.6 and 1.7 together with a well-known example in each class. Their taxonomy was more detailed than that of Flynn or Shore and took implicit account of pipelined structures. Therefore, the Multiple Instruction class was not considered for further categorisation as with the pipelined and array processor computers. Nevertheless, this scheme if coupled with that of Flynn could well be suited for a general classification of parallel computers.

-
1. Using a descriptive notation for differentiating between computers is much analogous to writing chemical formula. However unlike any chemical atom, a component, such as the execution unit, may have several different types (integer, floating-point, pipelined, bit-serial etc).

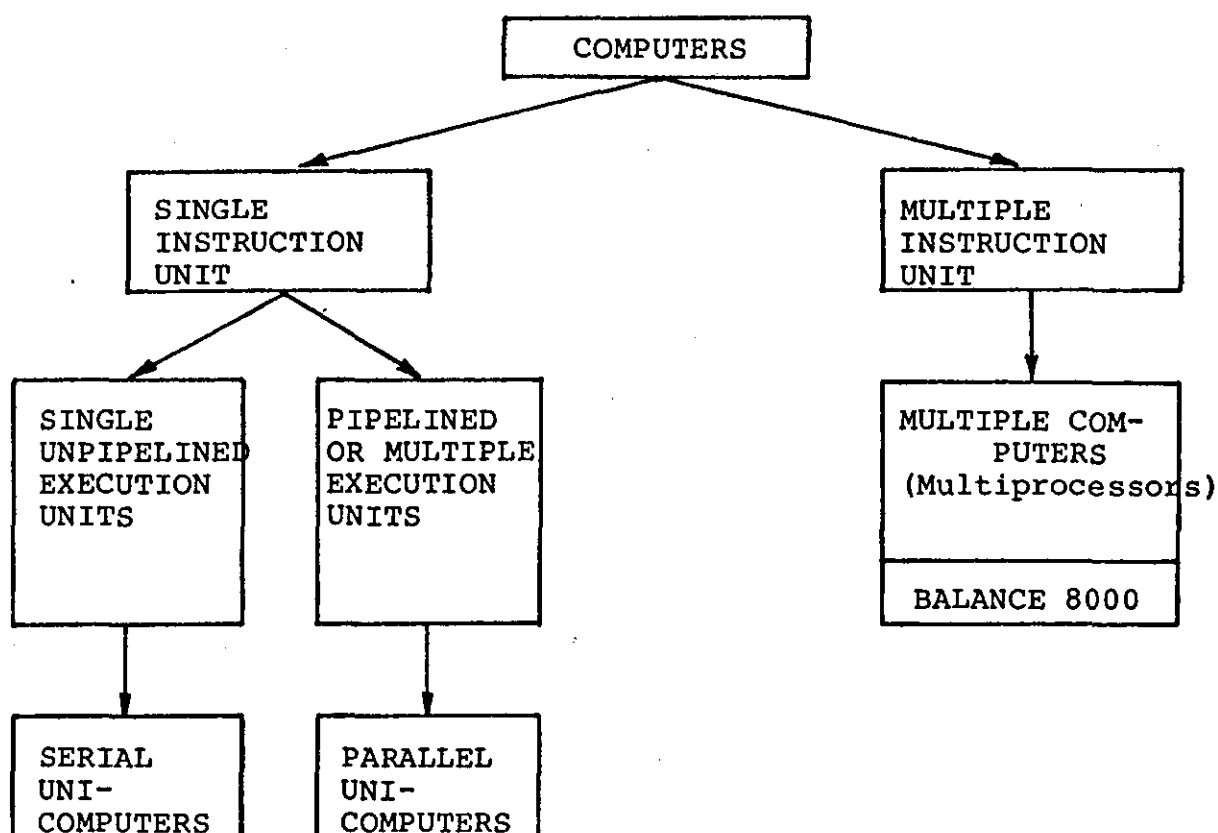


FIGURE 1.6: STRUCTURAL CLASSIFICATION OF COMPUTERS

1.4 PIPELINED COMPUTERS

The pipeline or vector notion, generally included in the parallelism notion, has been widely exploited since the 1960s when the need for faster and more cost-effective computer systems became critical.

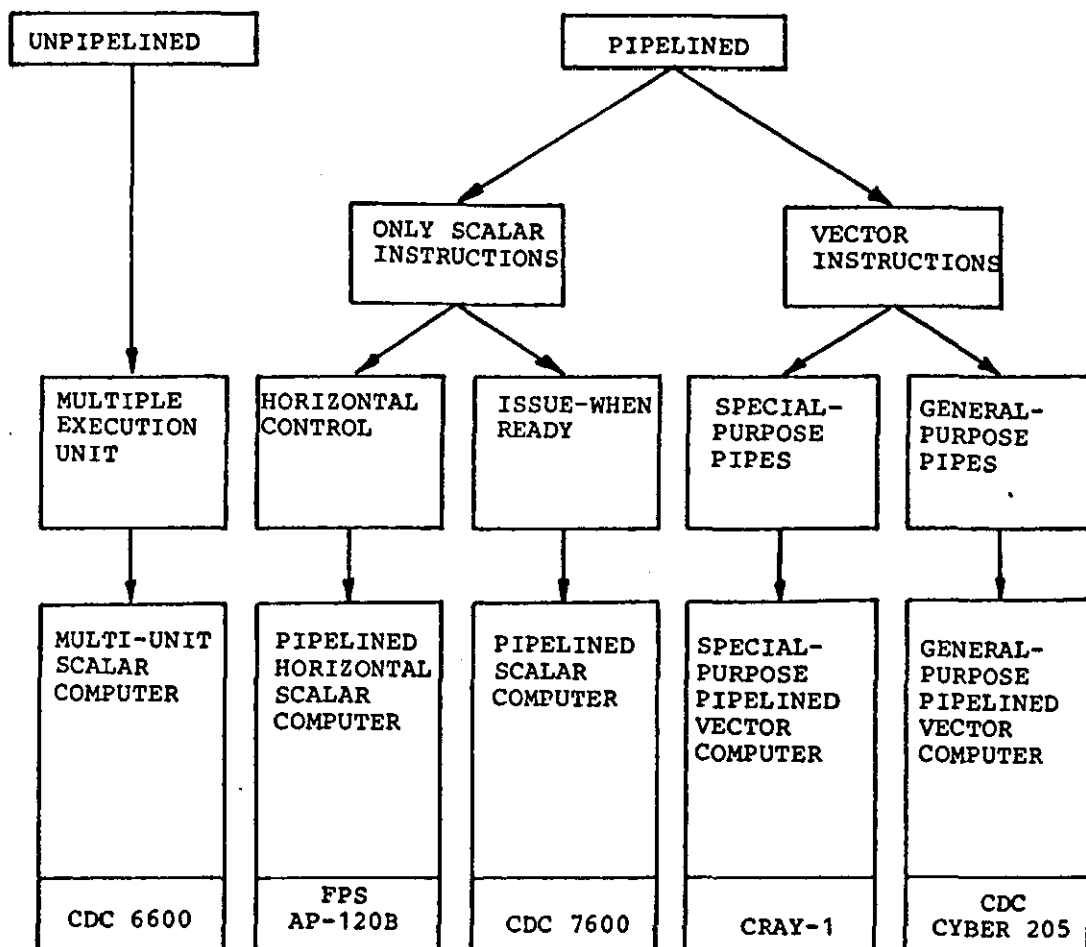


FIGURE 1.7: PARALLEL COMPUTERS BASED ON FUNCTIONAL PARALLELISM

Pipelining, a novel architectural design approach, is one form or technique of embedding parallelism or concurrency in a computer system. Although, essentially sequential, this type of computer helps to match the speeds of various subsystems without duplicating the cost of the entire system involved. It also improves system availability and reliability by providing several copies of dedicated subsystems.

In principle, the pipeline is closely related to an industrial assembly line. As in the assembly line, precedence is automatically observed, but it takes time to fill the pipeline before full efficiency per cycle is reached and time to drain the pipeline completely as the last trailing results are collected.

Figure 1.7 depicts the sequential and vector processing taxonomy derived from pipeline computers together with examples of some well known and commercially available computer systems. Although the pipelined computer architectures present somewhat different organisational characteristics when compared to SIMD and MIMD computer architectures, they are of significant interest because of the close connection between algorithms best suited for SIMD and those which achieve great performance on a pipelined computer system.

Machines such as the Texas Instruments Advanced Scientific Computer TI ASC [Watson 1972], CRAY-1 [Cray 1975] and the Control Data Corporation CDC STAR-100 [Hintz 1972] have distinct pipeline processing capabilities, either in the form of internally pipelined instruction and arithmetic units or in the form of pipelined special-purpose functional units. Ramamoorthy and Li [Ramamoorthy

1977] presented many of the theoretical considerations behind the pipeline notion and surveyed various pipelined computers that operate in either sequential or vector pipelined mode whose trade-off was studied. Also a top-down, level-by-level characterisation of pipeline applications in computers and the associated configuration control were explained in this reference.

The earliest use of overlapped modes of operation between the Central Processing Unit (CPU) and the Input/Output unit (I/O), namely an asynchronous input/output operation, can be found in the UNIVAC I, developed in 1951. This asynchronous Input/Output processing avoids having the processing unit waiting for the completion of I/O tasks and this improves the throughput. Within the processor, there can be an overlap between the Instruction Preparation or Processing (IP) and the Execution unit (E). The instruction preparation can be further subdivided into Instruction Fetch (IF), Instruction Decode (ID) and Operand Fetch (OF).

1.4.1 THE PIPELINE PRINCIPLE AND PERFORMANCE CHARACTERISTICS

Pipelined computers achieve an increase in computational speed by decomposing every process into several sub-processes which can be executed by special autonomous and concurrently operating hardware units. Furthermore pipelining can be introduced at more than one level in the design of computers. Ramamoorthy distinguished two pipeline levels, the system level for the pipelining of the processing unit and the subsystem level for the arithmetic pipelining. Particularly Handler [Handler 1982] introduced a third level and distinguished them under the names: macro-pipelining for the program level, instruction pipelining for the instruction level and the arithmetic pipelining for the word level. Others

distinguished the instruction pipelining, depending on the control structure in the system, to strict and relax pipelining. The flowing of the tasks through a strict pipeline is very rigid (must be smooth and ordered), whereas it is less restrictive in a relax pipelining (some turbulences in the data may occur from time to time) e.g. latter operations can move ahead of earlier ones.

In addition to the hierarchical levels of pipelining, a pipe can be further distinguished by its design configurations and control strategies into two forms; it can be either a static or dynamic pipe. Sometimes a pipelined structure is dedicated to a single function, e.g. a pipelined adder or multiplier. In this case it is termed unifunctional pipe with static configuration. On the other hand, a pipelined module can serve several different functions. Such a pipe is called a multifunctional pipe which can be static or dynamic depending on the number of active configurations (interconnections). If only one configuration is active at any one time, then the pipe is said to be static. Thus any overlapping of operations has to involve the same configuration. However, in a dynamic multifunctional pipe, more than one configuration can be active at any one time, thus permitting a synchronous overlapping on different interconnections.

The simplified model of a general pipelined computer is shown in Figure 1.8 where the processor unit is segmented into M modules, each of which performs its part of the processing and the result appears at the end of the M th segment.

The pipelined concurrency, a main characteristic of the simplest pipelining, is exemplified by the process of executing instructions. In Figure 1.9, we considered four modules: Instruction Fetch (IF),

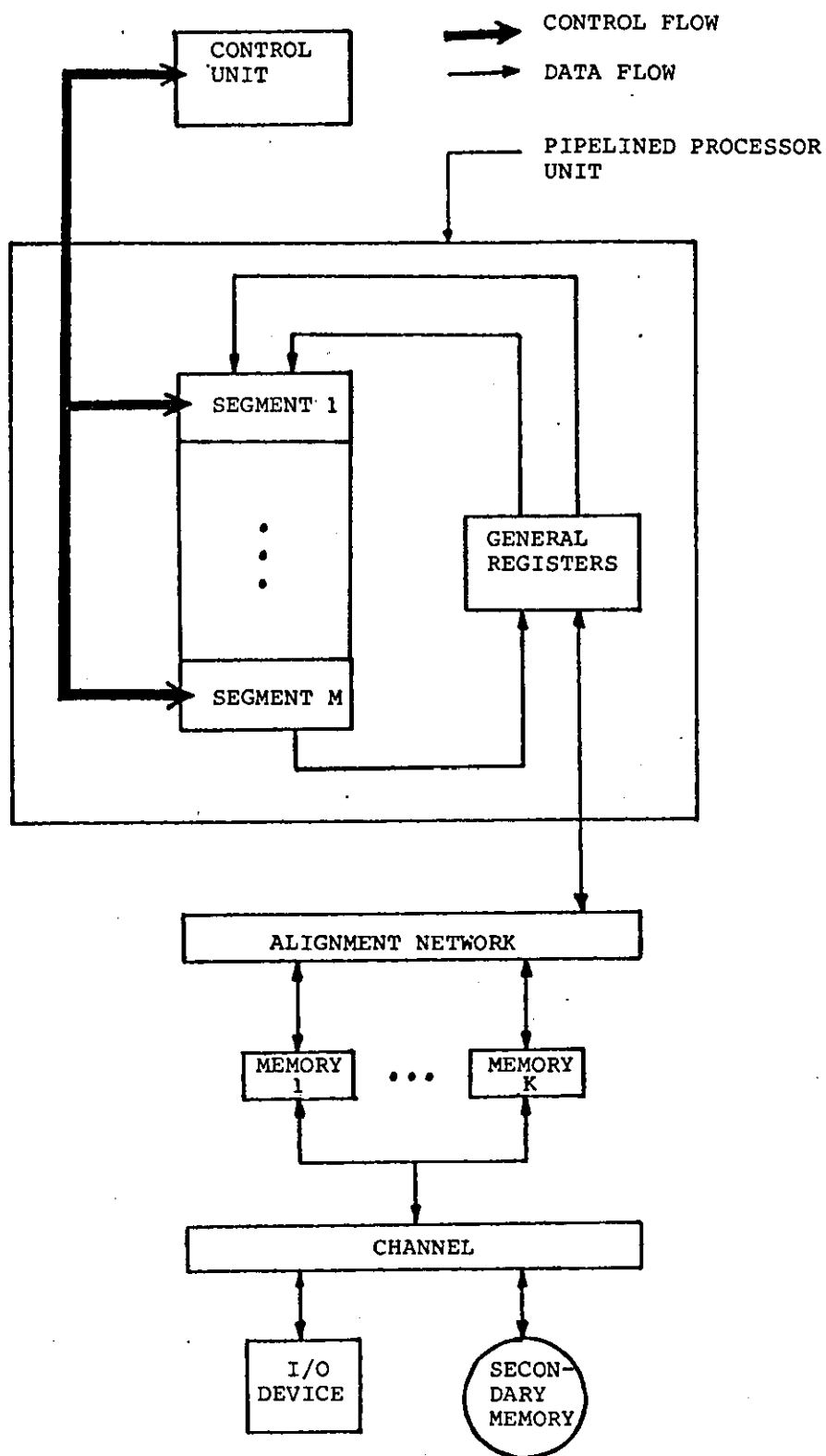


FIGURE 1.8: A PIPELINED PROCESSOR SYSTEM

Instruction Decode (ID), Operand Fetch (OF) and Execution (E), obtained when segmenting the process of processing instructions. Consequently, if the process is decomposed into four subprocesses and executed on the four-module pipelined system as defined above, then four successive instructions may execute in parallel and independently of each other but at different execution stages: the first instruction is in the execution phase, the second one is in the operand fetching stage, the third is in the instruction decoding phase and lastly, the fourth instruction is in the fetching stage. The overlapping procedure among these individual modules is depicted in Figure 1.10.

However the expected full-potential computation speed increase is not always achieved mainly due to some design and operational problems. These are buffering, busing structure, branching and interrupt handling. A brief discussion of these major design constituents along with the pipelining of the arithmetic functions is included. Their importance and effects which can actually decide the efficiency and performance of the resulting design are also outlined.

Buffering, an essential process to ensure a continuous smooth flow of data through the pipeline segments in the case where variable speed occurs, is virtually a process of storing the results of a segment temporarily before sending them to the next segment. Similar to an industrial assembly line, a segment may occasionally be slowed down for one of many reasons which could prevent the continuous input to the next station. To remedy this problem, a sufficient storage space or buffer is included between this segment and its predecessor, the latter can continue its operation on other results and transfer them to the provided buffer until it is full.



FIGURE 1.9: THE MODULES OF A PIPELINED PROCESSOR

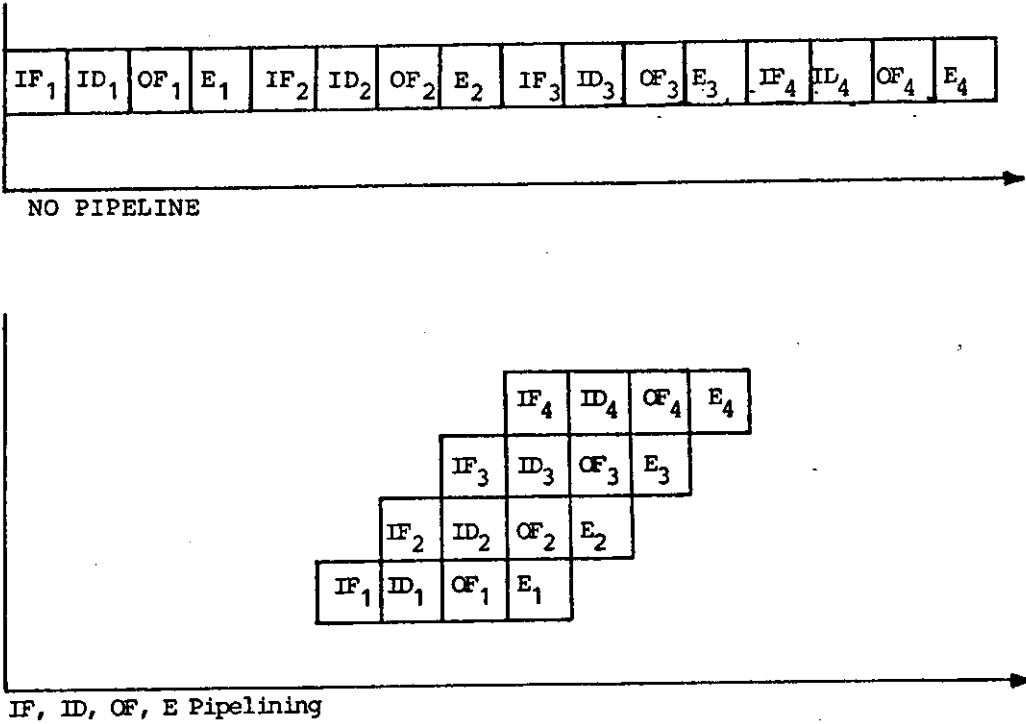


FIGURE 1.10: SPACE-TIME DIAGRAM

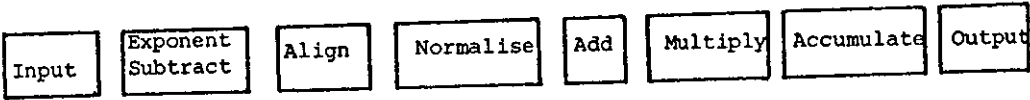


FIGURE 1.11: MODULES OF AN ARITHMETIC PIPELINED PROCESSOR

When the slowing down segment resumes normal service, it clears out its buffer, perhaps at a faster speed. Consequently buffering may be needed before and after a segment with variable processing time. The inclusion of buffering between segments in a pipelined structure makes the system perform at a relatively constant rate rather than at the speed of the slowest component. However full-speed is not always expected to be achieved since buffers have to be stabilized prior to any transfer activity.

In addition to the architectural features of the pipelined processor, the **busing structure** is equally important in deciding the efficiency of an algorithm to be executed on such a system. Pipelining, in essence, refers to the concurrent processing of independent instructions though they may be in different stages of execution due to overlapping. In real life, often, pipelined computers have to deal with **dependent or intermixed instructions**. With dependent tasks, their input and traversal through the pipe have to be paused before the dependency is tackled. The internal **busing structure** serves this purpose by routing the results to the requesting segments efficiently, thus reducing the adverse effect of instruction dependency, but still leaving a great burden on the programmer. However, in the case of intermixed instructions, more concurrent processing can take place since the resulting of dependency is hidden behind the processing of independent tasks.

Another damaging factor to the pipeline performance, even more than the instruction dependency is branching. The encounter of a conditional branch not only delays further executions but affects the performance of the entire pipe since the exact sequence of instructions to be followed is hard to foretell until the deciding result becomes available at the output. To alleviate the effects of

branching, several techniques have been employed to provide mechanisms through which processing can resume safely even if an incorrect branch occurs which may create a discontinuous supply of instructions.

A similar degrading effect to the conditional branching is caused by interrupts which disrupt the continuity of the instruction stream through the pipeline. Interrupts must be serviced before any action can be applied to the next instruction. In the case that the cost of a recovery mechanism for processing to proceed afterwards when an unpredictable interrupt occurs (while instruction *i* is the next one to enter the pipe), is not exceedingly substantial, sufficient information is saved for the eventual recovery. Otherwise these two instructions, the interrupt instruction and instruction *i*, have to be executed sequentially which is, in fact, not aimed at by the pipelining principle. An example of an interrupt recovery system is present in the STAR-100 processor in the form of special interrupt counters capable of holding important information such as addresses, delimiters, field lengths. These are necessary for the eventual recovery of vector-type instructions after an unpredictable interrupt has occurred. However, in a more general-purpose pipelined computer system the instruction recovery imposes a costly and complex problem. Also, different types of interrupt, depending on what they are associated with, can be distinguished. For example, in the IBM 360/91 two types of interrupts are used, namely, the **precise** interrupt, associated with an instruction (like an illegal instruction code) and the **imprecise** interrupt resulting from the storage, address and execution functions. The former type of interrupt occurs at the decoding stage whereas the second one might occur during other execution phases. In both cases, the next instruction to enter the decoding phase (let's say) instruction *i*,

is halted and all instructions present in the pipe (i.e. interrupted instructions) are allowed to complete execution before the processing unit is switched to service the interrupt.

Finally, one of the most beneficial applications of overlapped processing in order to increase the total throughput has been the execution of arithmetic functions. Specially, the advantages of pipelining are greatly enhanced when floating point operations are being considered since they represent quite a lengthy process. Again, until all modules in the pipe are excessively used, full speed is not obtained. For example, the TI ASC arithmetic pipelined processor is made up of eight modules, as shown in Figure 1.11.

Concluding, in order to determine whether a particular pipelined computer is efficient or not in terms of throughput, the following evaluation of the basic timings are performed. Suppose that in an idealistic situation, all sub-processes are designed to complete in time τ . In the case of a p -pipelined processor (containing p modules), then a process of at most p sub-processes requires $p\tau$ time units to complete and a succession of k such processes if overlapping is considered would be executed in $p\tau + (k-1)\tau$. The first result is output after $p\tau$ time units, time necessary to fill the pipe, and the $k-1$ remaining results are obtained at the rate of one result every unit time. If we define t as the time to complete a process in a sequential computer, then to achieve a faster execution time of the k consecutive processes we require

$$(k-1)\tau + p\tau < kt$$

$$k > (p-1) \frac{\tau}{t-\tau}$$

1.3.1.1

A fundamental conclusion to be drawn from the above inequality expresses the condition which, if satisfied, leads to a theoretical high throughput rate, namely the number of processes has to be long relatively to the number of modules in the pipe. In real applications, the throughput rate is also determined by its slowest segment or bottleneck. Techniques such as the subdivision of the bottleneck element or the multiplicity of this facility to perform in parallel (see Figures 1.12(a), (b) and (c)), are very useful in reducing the effect of bottlenecks. However, the latter technique is less appealing than the former since it also introduces more complex problems, namely the distribution and synchronisation of the tasks in the parallel portion of the pipeline.

FIGURE 1.12(a): THE BOTTLENECK IS IN MODULE 2

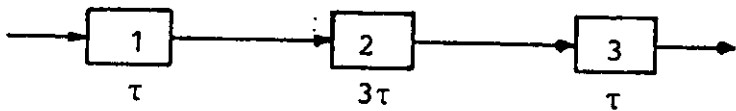


FIGURE 1.12(b): SUBDIVISION OF BOTTLENECK ELEMENT

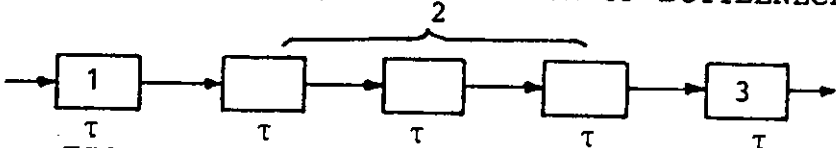
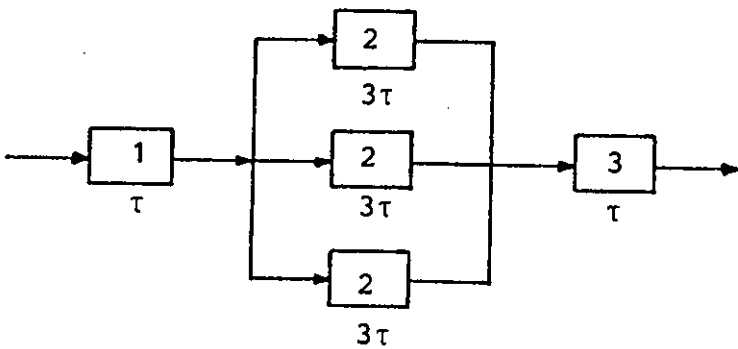


FIGURE 1.12(c): MULTIPLYING OF BOTTLENECK



1.4.2 VECTOR PROCESSING

Ideally, the throughput of a pipelined computer is maximised when a continuous excitation of the pipeline is frequently attained. This is equivalent to an almost continuous stream of independent instructions. Vector processing which represents a repetitive sequence of the same process on a set of different data, is a highly recommended process for pipelining. The overlapped characteristics of pipelining are employed when the required transformation of vector elements are independent of each other.

A vector pipe can be characterised by the existence of one or more multifunctional pipes in the execution unit (arithmetic and logic unit). In the case of a multifunctional pipelined execution unit, a static configuration can be established and retained throughout the entire vector processing. Hence, minimal control, decoding and reconfiguration overheads may be achieved while the memory operands are supplied to the execution modules in a most efficient way. Additional overheads such as the set-up time and the flushing-time are associated with vector processing. The former represents the time to fetch all the control and data parameters from the storage so as to structure the pipeline preparing the vector data streams and the latter overheads which directly measures the sum of the execution time of all facilities that the instructions and operand pair have to go through, is the period of time between the initial operation (the decoding) of the instructions and the exit of the result (for vectors, the first result element) through the entire pipe.

According to the above time constraints, the vector instruction processing time, t_{vp} , in the case of an effective vector field length k , can be expressed analytically as (assuming the bottleneck is in the execution units)

$$t_{vp} = t_s + t_{vf} + (k-1) t_e \quad 1.3.2.1$$

where t_s is the set-up time, t_{vf} is the flushing time including decode, address calculation, operand fetch and paired, termination check and execution, and t_e is the speed of the bottleneck segment.

Similarly, the execution of k operations in a sequential pipe, i.e. the same instruction has to be executed on a vector of data using a pipeline without vector processing power, can be analogously analysed. This instruction has to go through the entire pipe k times and thus the processing time can be expressed as:

$$t_{sp} = t_{sf} + (k-1) t_b \quad 1.3.2.2$$

where t_{sp} is the sequential (pipeline) processing time, t_{sf} is the sequential pipe flush time, and t_b is the speed of the bottleneck in the pipe. Comparing t_{vp} and t_{sp} yields

$$t_s + t_{vp} + (k-1) t_b \leq t_{sf} + (k-1) t_b$$

$$t_s + t_{vp} - t_{sf} \leq (k-1) (t_p - t_b) \quad 1.3.2.3$$

This last equation reveals that vector processing is beneficial when the length of the processed vector is considerably large; in other words, if the set-up and differential flush times are large compared to the difference of the speeds of the bottlenecks of the two pipes, then a lengthy vector field is required to justify vector processing.

Vector pipes are designed to be cost-effective, they are implemented with sufficient flexibility and power in order to match the speed of the Array Processors which are often more expensive.

In conclusion vector processing as compared with sequential pipelined processing, offers many advantages. In terms of time efficiency the speed is improved for considerably lengthy vectors, and in terms of resource utilisation, vector processing ensures a more efficient utilisation of all system facilities. The overhead incurred is principally in the additional software facilities required to use the pipeline efficiently. There is also a need for additional control circuitry, especially for multifunctional pipes, to establish the required configuration and routing of the data operands between pipe segments. Because of its cost-effectiveness and speed advantages, vector processing may be generalised and extended to smaller scale processing systems.

1.4.3 IMPLEMENTED PIPELINED COMPUTERS

As a concluding paragraph for this section on Pipelined Computers, we shall briefly present the architectural characteristics and performance of some of the commercially implemented pipelined

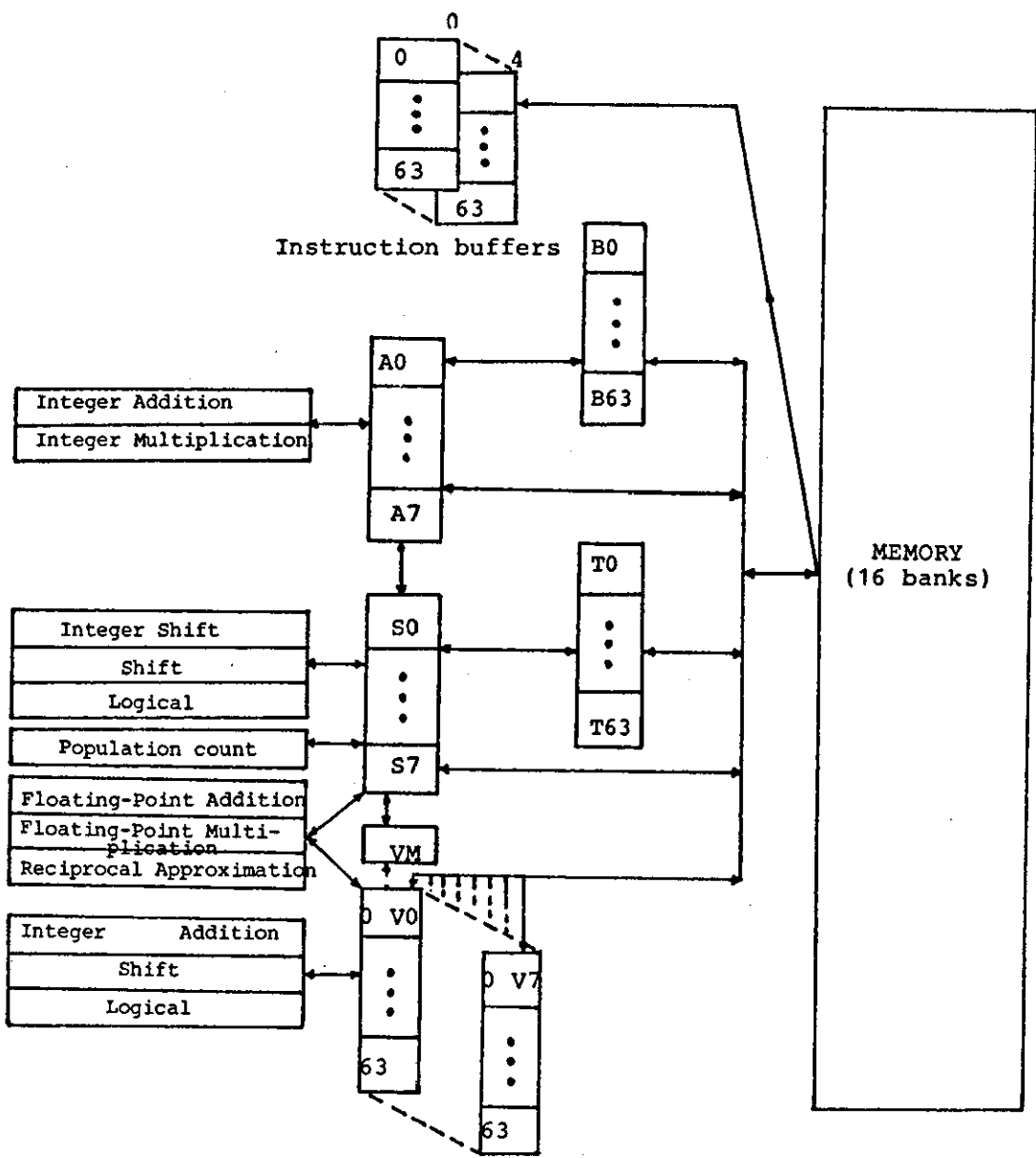
computers: CRAY-1, CDC CYBER 205, AMDAHL 470V/6, TI ASC and the FPS AP-120B.

The CRAY-1, manufactured by Cray Research Inc, at Chippewa, Wisconsin, USA¹, was the first successful Vector Pipelined Computer with a design philosophy following closely the CDC 6600 and 7600. One of the striking features of this machine is its small size: 4½ ft (feet) in diameter and 6½ ft high. Overall it comprises a main memory feeding data to or from a set of scalars and vector registers and twelve independent functional units to perform arithmetic and logic operations on the contents of these registers. (Figure 1.13 shows the main units and data paths of the CRAY-1). The maximum size of the memory on CRAY-1 is one million (exactly it is 2^{20}) 64-bit words of bipolar memory with 50 ns access and cycle time, divided into 16 memory banks that may operate simultaneously giving a maximal bandwidth of 320 Mword/s (million words per second). This has been increased to 4 Mwords on the CRAY-1S which was announced in 1979. There are four instruction buffers each holding 64 16-bit instruction words and each is connected to memory by a 64-bit wide data-bus which can achieve a transfer rate of four instruction words per clock period (a bandwidth of 320 Mword/s). These maxima apply only when all the instructions are drawn from separate memory banks.

The data-bus between the registers and main memory, on the other hand, is only 16-bit wide allowing a low data transfer rate of only 80 Mword/s. The maximum computing rate on the CRAY-1 is 160 Mflops/s (80 million multiplications and 80 million additions per second) with a clock period of 12.5 ns.

1. The first delivery was made to the Los Alamos Scientific Laboratory, New Mexico in February 1976.

FIGURE 1.13: ARCHITECTURAL BLOCK DIAGRAM SHOWING THE PRINCIPAL UNITS, BUFFERS AND DATA PATHS OF THE CRAY-1 COMPUTER



The registers use 6 ns logic and comprise eight 24-bit address registers (A0 to A7), eight 64-bit floating point scalar registers (S0 to S7), and eight floating point vector registers (V0 to V7) each of which can hold up to 64 64-bit floating point numbers. Sixty-four buffer registers are also provided between the A registers and main memory (24-bit registers B0 to B63) and between the S registers and main memory (64-bit registers T0 to T63). The main memory may send data either to the A and S registers at the maximum rate of one word every two clock periods (40 Mword/s), or to the B, T and V registers at the maximum rate of one word per clock period (80 Mword/s). The buffer registers are used for the purpose of storing intermediate results which may be transferred to the A and S registers in one clock period.

The design of the CRAY-1 was mainly motivated by the commercial desire to provide a substitute for existing computers such as the CDC 7600 and the IBM 360/195 to be sold to major scientific research centres. Unlike the ILLIAC IV which pioneered the development of several technological innovations, there was no incentive for experiments with new technologies. In fact the technological choices in the CRAY-1 were therefore conservative and the novel of the machine appears mainly in the architecture.

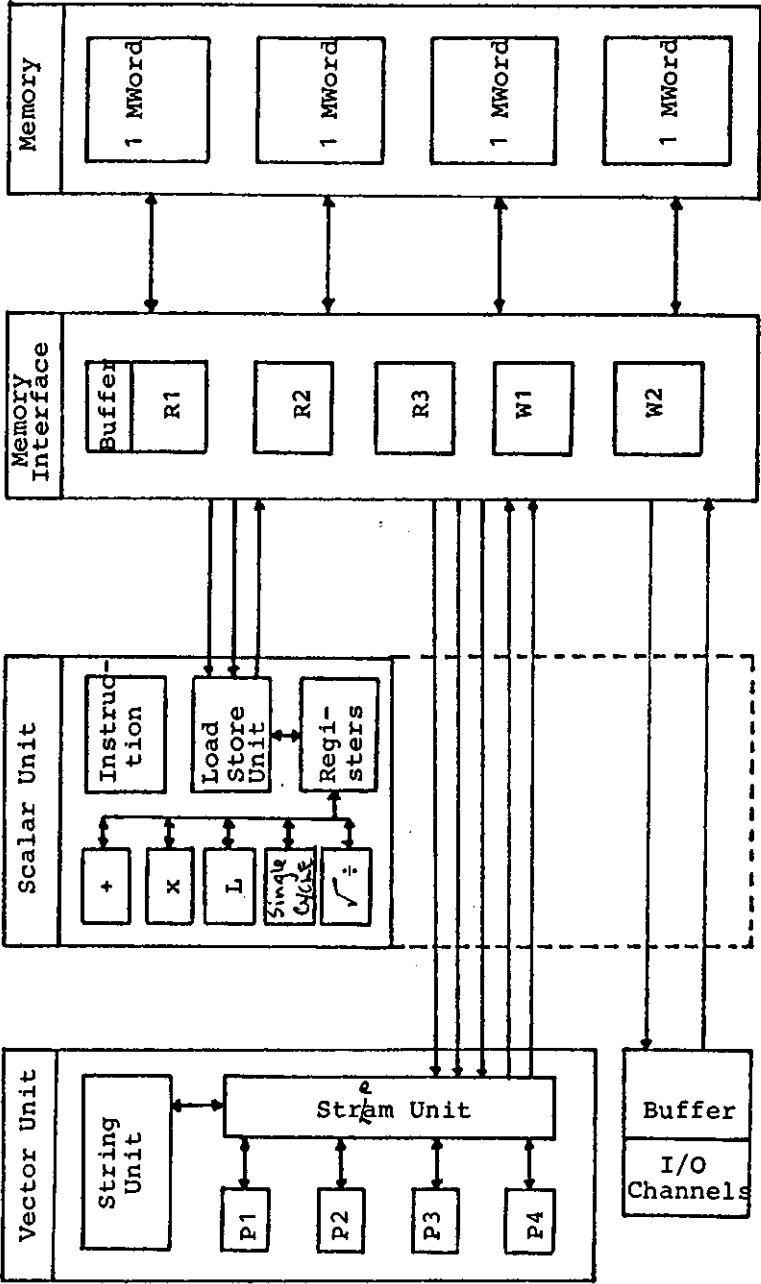
The CDC CYBER 205¹ manufactured by Control Data Corporation in Saint Paul, Minnesota, USA represents the culmination of a long program of research and development that started with the design and delivery of the CDC STAR 100 computer in the period 1965-75. When

1. It was announced in 1980 and the first delivery was made to the UK Meteorological Office, Bracknell, England, in 1981.

operational, the CDC STAR 100 showed many disadvantages which made it unattractive to potential customers. Consequently, Control Data Corporation decided to develop a new LSI technology and re-engineer the whole system using it, as well as to make a few improvements in the architecture. One of the decisions taken at that time was to retain the use of the extensive developed software by retaining the STAR 100 instruction set and overall organisation. The re-engineering program also included the means of upgrading existing STAR 100 machines to that of the re-engineered version. This took place in two stages:

Firstly, the CYBER 203, also known as STAR 100A, was manufactured and announced in 1979 by re-engineering the STAR 100 structure whose slow main memory was replaced by a 80 ns tripolar memory. The scalar and short-vector performance was also boosted by the addition of a new developed LSI scalar unit with a 20 ns clock period. However, the two vector pipes and the stream unit of the STAR 100 with a 40 ns clock were unchanged. The second stage was the manufacture of the CYBER 205 (initially known as the STAR 100C and subsequently as the STAR 100E) which is the CYBER 203 with re-engineered and improved LSI vector pipes and stream unit, working with a 20 ns clock period. The CYBER 205 machine offers several architectural options (see Figure 1.14). The number of pipes may be optionally increased from two to four, the memory to 4 Mwords and the I/O channels to 16. However, the disadvantage of a unit vector increment caused by the contiguous vector requirement, i.e. successive vector elements should be stored in successive memory locations, remained. The maximum overall performance of the CYBER 205 is 800 Mflops/s in 32-bit arithmetic on a four pipe machine.

FIGURE 1.14: ARCHITECTURAL BLOCK DIAGRAM SHOWING THE PRINCIPAL UNITS AND DATA PATHS OF THE CDC CYBER 205 COMPUTER



The AMDAHL 470V/6, an IBM 360 compatible computer manufactured by the AMDAHL Corporation, was the first computer to use LSI in the logic circuits of the CPU. The first delivered machine in 1975 had a basic cycle time of 32.5 ns which was reduced to 29 ns in the subsequent version of the AMDAHL 470V/7. Although the arithmetic units in this machine were not pipelined, a high performance of 4.6 Mflops/s was obtained by pipelining the processing of instructions which was organised into 12 sub-operations. When flowing smoothly, an instruction could be taken every 2 clock periods, therefore up to six instructions could be in parallel executions. Also, a high-speed buffer (or cache) bipolar memory of 26 Kbytes with a cycle time of 65 ns was used to improve the effective access time of the slow main memory of up to 8 Mbytes of MOS store (650 ns access).

The TI ASC system, manufactured by Texas Instruments, was started around 1966 as a computer suitable for the high-speed processing of seismic data. It is based on four identical general-purpose pipelines, each capable of performing the elementary instructions on vector operands. Instructions can be taken from one or two instruction processing units which, when operating in parallel, improve the instructions throughput. (Figure 1.15 shows the architecture block diagram of the TI ASC computer). With the four pipes operating optimally, a design rate of 50 Mflops/s was achieved. The semiconductor memory had eight banks and a cycle time of 320 ns. After installing about seven systems, the first one was in 1973, the manufacture of the TI ASC was discontinued because, like the STAR 100, it also suffered from a scalar unit that was significantly uncompetitive.

Concluding this paragraph, the FPS AP-120B, the first low-cost yet high-performance computer manufactured by Floating Point Systems

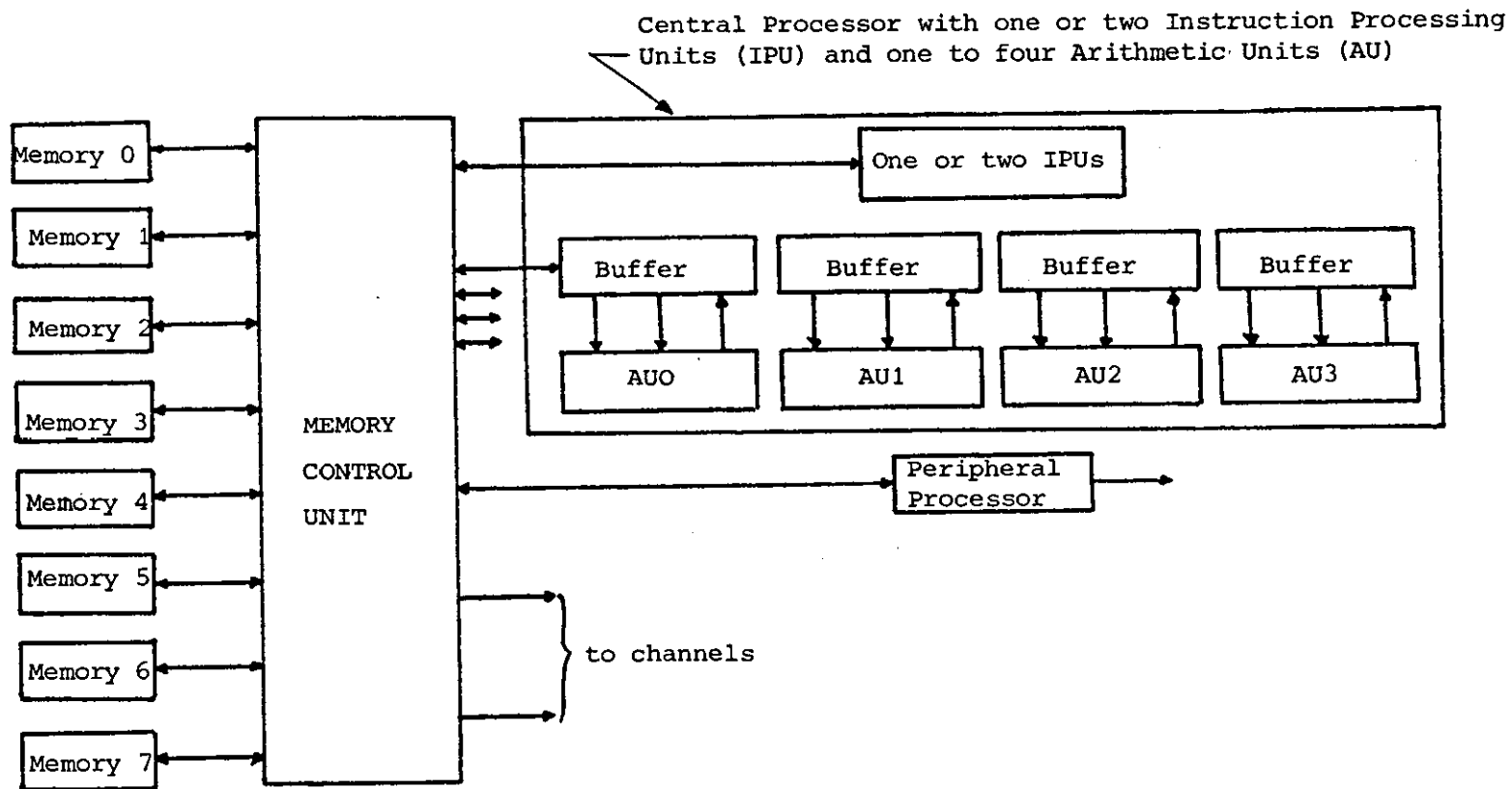


FIGURE 1.15: ARCHITECTURAL BLOCK DIAGRAM SHOWING THE PRINCIPAL UNITS AND DATA PATHS OF THE T1 ASC COMPUTER

Inc, is called Array Processor (AP)¹, since it performs efficiently on arrays of numbers. The first deliveries started in 1976. Comparing the overall architecture and the physical layout of the AP-120B and CRAY-1, one might conclude that the former is to a mini or a medium computer, what the CRAY-1 is to a large main frame computer. The overall architecture (see Figure 1.16) of the AP-120B is based on multiple special-purpose memories feeding two floating-point pipelined arithmetic units via multiple data paths. It is driven synchronously from a single clock with a period of 167 ns. The standard memory has an access/cycle time of 500 ns, whereas the optional fast memory has a cycle time of 333 ns. Both types of memories are, however, organised as a pair of independent memory banks (one for the odd addresses and one for the even addresses). Depending on the type of memory, standard or fast, successive references to the same bank must be separated by at least 3 or 2 clock periods respectively whereas they can be made on successive clock periods when referring to different banks. The system includes a 38-bit floating point arithmetic unit organised into two separate pipelined multiplication and addition units and an independent 16-bit integer arithmetic unit for counting and address calculation. Three memories (program memory, data memory and table memory) and 2 scratch pads of registers (X and Y registers) are provided with multiple data paths between each memory bank and each functional unit. Theoretical processing rates of 5-10 Mflops may be achieved.

1. The fact that the machine is called an array processor does not mean that it is composed of an array of processors.

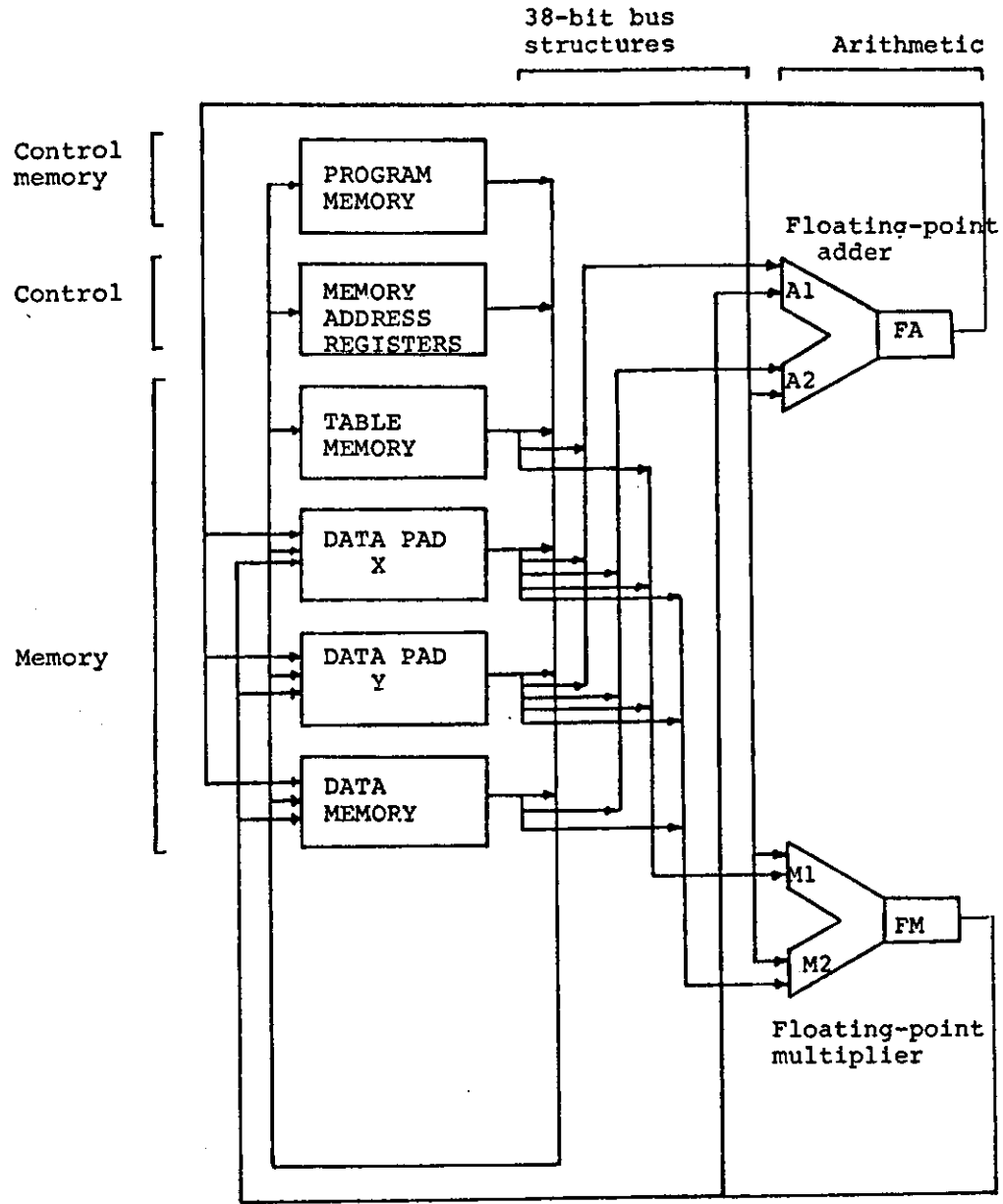


FIGURE 1.16: ARCHITECTURAL BLOCK DIAGRAM SHOWING THE PRINCIPAL UNITS AND DATA PATHS OF THE FPS AP-120B COMPUTER

A successor prototype to the AP-120B was announced in 1980 under the name of AP-164¹ with first deliveries in 1981. The principle improvements introduced which maintained the overall structure, the clock rates and machine timings were a 64-bit floating-point arithmetic, a 32-bit integer arithmetic, a 24-bit addressing, a 1024 64-bit word instruction cache memory loading from the main memory, replacing the program memory of the AP-120B and a main memory expandable to 12 Mbytes.

Finally, the arrangement of control in the AP-120B computer was referred to as horizontal microcode since each 64-bit wide instruction controls all the functional units every clock period. Thus there is only one instruction with fields controlling each of the functions, although certain combinations of functions were excluded due to some fields overlapping.

1.5 DATA-FLOW COMPUTERS

A common feature for all the high-speed parallel computer architectures is that, due to the basic linearity of the program, the use of implicit sequencing of the instructions is possible. This is a Von-Neumann characteristic which means that the order of execution of the instructions is determined by the order in which they are stored in the memory with branches used to break this implicit sequencing at selective points.

1. This computer was preceded by the AP-190L which is an enhanced version of AP-120B, with more memory.

An alternative form of instruction controlling is the explicit sequencing which is basically the principal concept exploited by the data-flow machines to provide the maximum possibilities for concurrency and speed-up. However, this concept has a significant impact on the architecture of such machines, the program representation, and the synchronisation overheads.

In a data-flow architecture the algorithm is represented by a graph where the nodes correspond to the computations and the arcs describe the flow of data or operands, from the node producing the data (as a result) to the node using it as an operand [Dennis 1980]. In addition to the nodes describing the basic operations, there are nodes which are used to control the routing of data. Thus, the execution of any instruction is determined by the availability of all its operands resulting in a more complex control due to the high overheads involved in routing the data.

With the use of the above graph representation, the data-flow concept encounters some problems when the algorithm contains loops or subroutine calls, in which case the same instruction is executed several times. Basically, the implementation of the data-flow computers can be grouped into two main classes, the static and dynamic structures, depending on how this problem is tackled. In the first class, the static one, the loops and subroutines calls are unfolded at compile time so that each instruction is executed only once. Consequently, the implementation of the sequencing control is made simple since it directly follows that of the graph. On the other hand, in the dynamic case, the operands are labelled so that a single copy of the same instruction can be used several times for different instances of the loop (or subroutine). Since, in this type of architecture, it is necessary to match all the operands with

the same label before issuing the single copy of the instruction, the implementation of the control is significantly more complex in comparison with that of the previous class. However, the dynamic approach which allows a compact representation of large programs, can effectively exploit the concurrency that appears during execution (for example, recursive calls or data-dependent loops).

An example of the static approach is the MIT Data-Flow Machine (see Figure 1.17) which consists of the following main components: a store that contains the instruction cells or packets having space for the operation, operands and for pointers to the successors, and a set of operating units to perform the operations. These two components are connected by two interconnection networks, one to send ready-to-execute instruction packets to the operating units and another to send results back from the operating units to the instructions that use them as operands. The system has to be carefully designed so as to prevent any bottleneck from occurring and to provide means for the full exploitation of all the concurrency.

In such a system, the maximum throughput is determined by the speed and number of the operating units, the memory bandwidth and by the interconnection system. As in the other organisations, several degradation factors reduce the effective throughput. The most significant are the degree of concurrency available in the program, the memory access and interconnection network conflicts, and the broadcasting of results, all of which except the last one, are similar to other systems. Sometimes an instruction has several successors, so that the result has to be sent, or broadcasted, to all of them and this introduces significant overheads in the case when the number of destination pointers present in an instruction

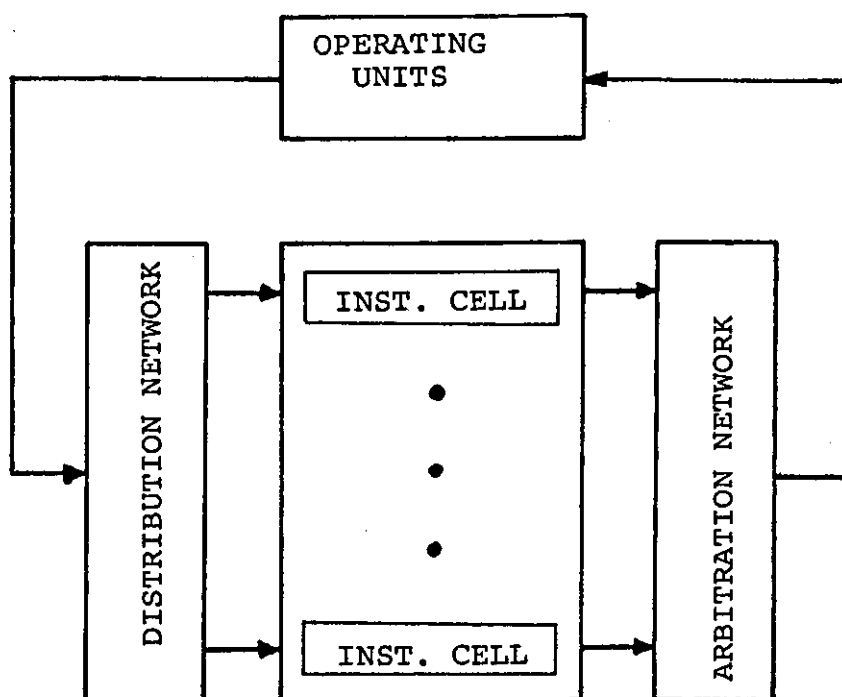


FIGURE 1.17: THE STATIC DATA-FLOW MACHINE

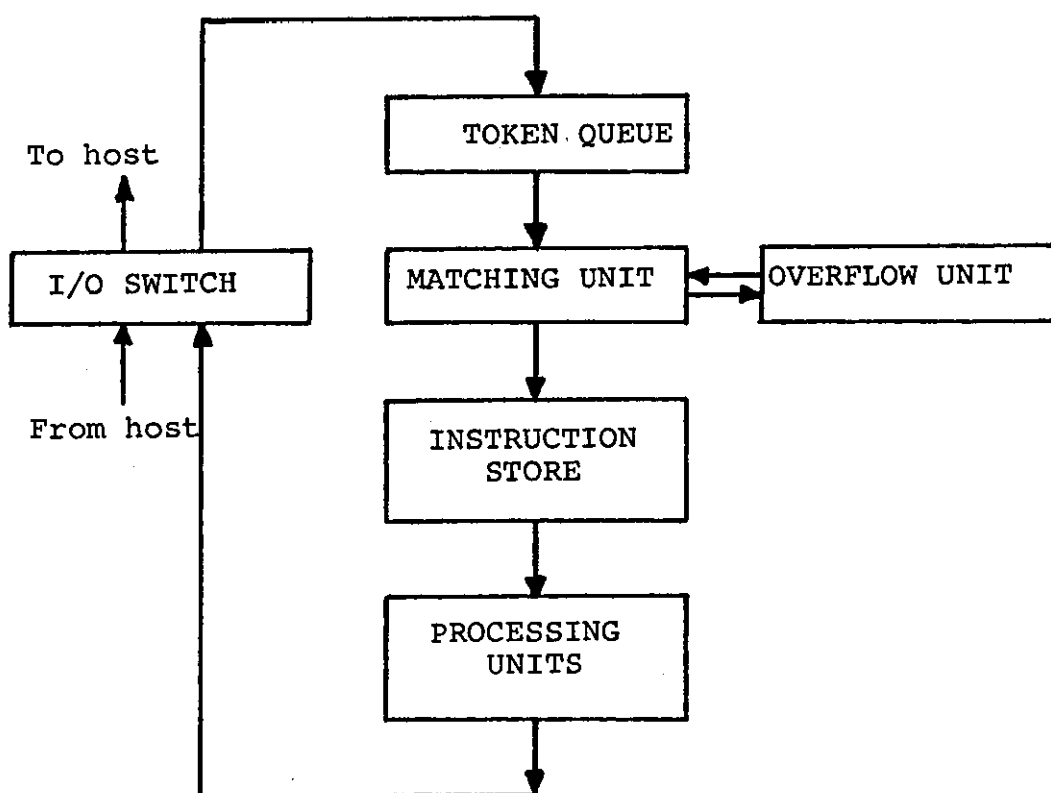


FIGURE 1.18: MANCHESTER DATA-FLOW MACHINE

cell is limited.

Examples of the dynamic approach include the U-Interpreter Machine [Arvind 1982] and the Manchester Dataflow Machine [Gurd 1985]. The main components of the latter (see Figure 1.18) are the token queue that stores computed results, the token matching unit that combines the corresponding tokens into instruction arguments, the instruction store that holds the ready-to-execute instructions, the operating units, and the I/O switch for communication with the host. The degradation factors are similar to those of the static case except the additional overhead in token label matching. Due to the above mentioned degradation factors, data flow machines are only attractive for cases in which the concurrency exhibited is of several hundred instructions.

Another problem in the use of the dataflow approach is the lack of any data structure definition, in fact only scalar operations were first utilised in the attempt to maximise the amount of concurrency and this had significant limitations in terms of the modularity of the programs. The inclusion of data structures in the graph representation requires that the dataflow concept be extended and operations on them be defined [Davis 1982]. From the operational point of view, the most straightforward solution is to treat the data structure as an atomic operand, requiring the structure to be sent as a whole to the operating units even though only few elements are operated on. This can be performed by sending to the operating unit a pointer to the data structure instead of its value. However, the disadvantage with this is that the whole data structure has to be copied when any of its elements is modified resulting in a heavy transfer rate between the memory and the operating units. To avoid this copying overhead, Dennis [Dennis

1974] has proposed a tree structure to store arrays and operations such as select and append to modify parts of the array. However, Dennis' proposal does not solve the limitation that the elements of the array have to be modified in a sequential manner, which increases the overhead for the select and append operations. To reduce this overhead Gandiot and Ercegovic [Gandiot 1982] proposed the introduction of macro-actors to perform more complex updating. To eliminate the sequential nature of the modifications, Arvind and Thomas [Arvind 1980] introduced I-structures that allow concurrent writes and reads by adding to each element a tag indicating if the element has already been written, and a list of pending reads to the reads queue to arrive before the element has been written.

One of the most significant advantages of the dataflow machines, as claimed by its proponents, is the exploitation of the concurrency at a low level of the execution hierarchy since it allows the maximum utilisation of all the available concurrency. However, some researchers argued that the overhead with this unstructured low-level concurrency is too high and have proposed the use of a hierarchical approach in which different types of concurrency can be exploited at different levels.

Finally, the dataflow organisation which is still in an experimental stage, has recently received considerable researchers' attention. Several prototype systems have been built or simulated and are being evaluated. Some examples are the MIT dataflow machines (The Static Data-Flow Machine [Dennis 1983] and the Tagged-Token Data-Flow Machine [Arvin 1983a]). The Manchester Data-Flow Machine [Gurd 1985], the TI Data-Flow Machine [Cornish 1979], and Single Assignment Data-Flow Machine LAU [Comte 1980].

Chapter 2

CURRENT MULTIPLE PROCESSOR SYSTEM ARCHITECTURES

2.1 INTRODUCTION

The rapid development of computer technology has not only produced high performance processing systems but has had a significant impact on its terminology. More specifically, the terms **parallel computers** and **parallel processing**, as used in these first chapters, refer to the early computers performing arithmetic operations on whole words, rather than on a single bit, and continues right up to the more recent notions of **multiple-processing** or **concurrency** as presently implemented in several commercial and experimental high-speed parallel computers.

In this chapter, we shall consider the SIMD and MIMD type of computers separately and present in a detailed fashion, the hardware and software characteristics that must be possessed by the respective architecture in order to be classified as such. In addition, both classes are exemplified by a detailed architectural discussion of some of the well known implemented computer systems. For the SIMD class, a further sub-categorisation is undertaken to produce two subclasses, the **Associative** and **Array** processor computers, which are thoroughly examined in Sections 2.2.2 and 2.2.3 respectively.

Although, we have referred to many computer classification schemes which have yielded several different types of computers, we shall not present all the conceivable parallel computer architectures, but only those which have contributed most to the achievement of many of the proposed design goals. However, due to the impact of VLSI circuits, the first two chapters will be complemented by a discussion of some VLSI architectures (see Chapter 4) in order to

provide a survey of most of the high-speed parallel computers of today.

2.2 THE GENEALOGY OF THE SIMD COMPUTERS

The main characteristics of the SIMD (Single Instruction Multiple Data Stream) category of computers is a single global control unit that drives more than one processing element, all of which can either execute or ignore the current instruction. Consequently these machines are known as **synchronous** computers since all processors would execute the same instruction at the same time but on separate data streams.

Following Flynn's classification of large-scale high-speed computer Thurber and Wald [Thurber 1975] gave some generic relationships which were subsequently amended and used in the creation of the SIMD sub-classifications. Accordingly, three sub-classes for the SIMD category were established: the **Associative Processors**, the **Parallel or Array Processors** and the **Ensembles**.

Associative processors which will be briefly described in Section 2.2.2, are characterised by the use of the **Associative Memories** instead of the conventional location-addressed memories and by the search capabilities offered by these memories. Section 2. .3 describes the architectural features of the Array Processors and the interconnection networks which are fundamental to the Array Processor Design and equally important to their efficient use. This section also includes a review of some implemented Array Processor Systems. In the **Ensemble** architectures, the least interesting architectures among the three SIMD sub-classes of computers, the

interconnection level between the processors could be non-existent or very low.

Finally, it should be understood that SIMD computers are special-purpose processors which require a front-end host computer to be attached to and thus they are only useful for a limited set of applications. The utilisation of SIMD computers, in general, and their applications are summarised in the following section.

2.2.1 THE UTILISATION AND APPLICATIONS OF THE SIMD SYSTEM

As with any special-purpose computer, the potential high performance can be achieved only when it is used properly (i.e. they are utilised for the purpose they were designed for) and SIMD systems are no exception. The most important aspects for utilising SIMD computers can be classified into the following three classes.

The first class, characterised by the hardware structure of these systems, is more efficient, compared to other multiprocessors, for problems with large amounts of parallelism provided that the cost of duplicating structures is not substantially high. Especially, with the advent of LSI microprocessors, the scope of SIMD utilisation will greatly increase as a result of a considerable drop in component costs.

The characterisation of the second class depends on the software specially designed for these systems and which tends to be simpler due to less executive function requirements than that needed in multiprocessors. As a result of the software simplicity, the construction of large systems is made even easier.

Finally, the functional utility of these systems which comes in the third class proves to be more efficient than Multiprocessors for large problems requiring intense data processing, e.g. weather forecasting, and for problems with inherently global parallelism; thus providing at the same time reliability, simpler system complexity and higher computational throughput.

On the other hand, the categorisation of the problems on which a cost effective implementation of the SIMD systems may be possible, has concentrated all the scientists' research attention. However solving a problem does not always mean just offering a problematic solution. It also includes a detailed description, a complete analysis of the nature of the problem and, most importantly, the actual implementation, which must take into account the economical benefits of the problem. This means that attention should be paid to the systems aspects too when developing solutions for problems.

Several scientific applications such as those for matrix manipulations, differential equations and linear programming have been proposed for Associative and Array Processors. It is often the case that applications, suitable for one type of computer, are not always the best for the other type.

The utilisation of the SIMD computers has made one thing clear. The elimination of critical bottlenecks, which had appeared in the general-purpose computer systems and affected their performance, was possible. Technological and economical problems had constrained the early Associative Processors to use only small associative memory systems (up to 1 Kwords with bandwidth up to 100-bit wide) for numerous but limited size problems such as computer resource management, and allocation etc. However, due to the development of

new architectural concepts and the use of LSI technology associative memories became larger and more flexible, thus putting Associative Processors to practical use.

Summarising the numerous applications that the SIMD systems are well suited for, with a distinction between Associative and Array Processors would be the following: air traffic control, radar tracking, bulk filtering and signal processing are some of the applications to be cost-effectively implemented on Array Processors. Similarly, for the Associative Processors, but bearing in mind the cost-factor constraint, the applications include resource allocation, virtual memory mechanism, interrupt processing protection mechanisms and scheduling.

However, in the case where the cost-factor constraint is not important, the above applications list for the Associative Processors could also include sorting, pattern recognition fields, sea surveillance, picture processing, graph processing, differential equations, eigenvectors, matrix operations, network flow analysis, data file manipulations and searching, compilation, theorem proving, computer graphics and weather forecasting. While some of the above applications are suited for the Array Processor, they are best for the Associative processors because of the search capabilities offered by the Associative memories.

Finally, we should mention that Slotnick [Slotnick 1967] and Fuller [Fuller 1967] have separately carried out a comparative study of Associative and Parallel Processors. They both concluded that Parallel Processors, as special purpose computers, appeared to be more useful than the Associative Processors.

2.2.2 ASSOCIATIVE PROCESSORS

The fundamental concept behind the Associative Processors basically depends on the extensive search capabilities offered by the associative memories which are most efficient for non-numerical applications such as radar signal tracking and processing, weather prediction computations and many types of information processing etc.

Due to the cost-factor constraint, Associative Processors are usually designed as a back-end, or special-purpose processor attached to a conventional sequential computer system. Thus problems requiring a high amount of processing power and which cannot be solved efficiently on the host computer are transferred to the Associative processor system.

Slade and McMahon [Slade 1957] were the first to develop associative memories by using cryotrons. Since then, many other different components such as magnetic cores, semiconductors, magnetic films and integrated circuits etc, were used in the construction of associative memories. The first Associative Processor was designed by Behnke and Rosenberger [Behnke 1963] in 1963 using cryotrons, against all the then existing constraining factors such as the high implementation costs, the half-select noise limiting the word width and the interrogation drive problems limiting the size of the associative memory (number of words).

Associative Processors were not practical until the development of the PEPE - 'Parallel Element Processing Ensemble' (see [Crane 1972], [Wilson 1972], [Cornell 1972], [Evensen 1973], [Dingeldine 1973] and

[Vick 1973]) where the use of LSI components and newly developed architectural concepts had broadened the bounds of the associative memories. Other Associative Processors such as the STARAN (see [Rudolf 1972], [Batcher 1972] and [Davis 1974]) and OMEN - 'Orthogonal Mini-Embedment' (see [Higbie 1972]) were also developed as a result of the LSI evolution.

Finally, the two main properties characterising this class of Associative processor system are emphasised. The use of a dedicated associative memory which retrieves stored data items using their content or part of it and not their addresses and the provision of multiple processing units capable of simultaneously performing the same data transformation, either arithmetic or logic, but on different data items. From the high processing rate point of view, content-addressed memories have contributed to the superiority of the Associative Processors when compared to the traditional sequential processors. Consequently, problems like weather forecasting, and the handling of large database requiring a huge amount of processing time and which cannot be run efficiently on sequential processors, are tackled faster and easier.

The architectural block diagram of Figure 2.1 shows the principal units of a general Associative Processor. They are an associative memory, an arithmetic and logic unit, a control system, instruction memory and an Input/Output interface. Due to the impact that the associative memories have had on the architecture of the Associative Processors, a classification of this class of processor, based on the associative memory organisation, is possible. First, a brief description of the memory organisation is outlined in the following section.

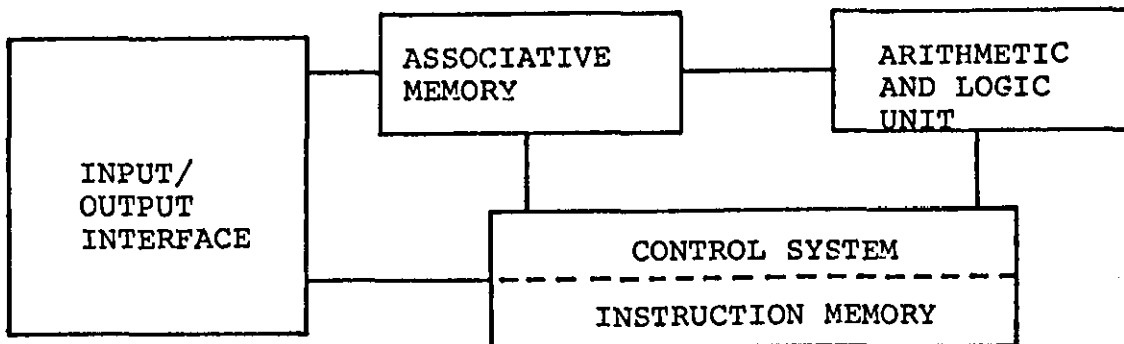


FIGURE 2.1: GENERAL BLOCK DIAGRAM OF AN ASSOCIATIVE PROCESSOR

2.2.2.1 ASSOCIATIVE MEMORY ORGANISATION

In the literature, associative memories were known under several names such as catalog memory, content-addressed memory, data-addressed memory, parallel search memory, search memory, search associative memory, distributed logic memory, associative push down memory and multi-access associative memory. Basically, these types of memories differ from the conventional memory systems by the fact that stored data items are addressed by their content or part of it, instead of their implicit location (address).

The memory search method which depends on the organisation of the implemented associative memory consists of two basic operations, namely masking and comparison. The comparison between a preset search-key word and all the words in the memory which can be performed either bit-parallel or bit-serial is achieved through the

interrogating bit drives and the available logic circuitry. A word-match tag network flags the multiple matched words which can be retrieved at the end of the searching procedure using a single instruction.

The above associative memory search operation is further illustrated by considering for example a personnel file of a computer data processing centre.

Imagine that at one stage of a query transaction information about all employees with a salary in the range £800 - £1600 inclusive per month has to be searched. This can be done using a **greater than** and a **not greater than** search operation, each of which can be performed in parallel, on the salary field of the file. Since the search is only concerned with the salary field, a mask is used to mark all other fields. Also, a match-word indication is required to indicate the results of the search. A single bit, associated with every word in the memory is used for this purpose where a value of 1 indicates a match and 0 otherwise.

More specifically for our example, the search-key word is loaded with the salary figure (800) and the indicator value (0) for the logical operation **greater than**. Every word in the associative memory would have its indicator field set to zero. After the first search, all matched words are memorised by having their indicator bit field set to one. Similarly, the second search-key word will have been loaded by the interrogating salary figure (1000) and the indicator (1) and the logic operation involved is **not greater than**. After the second search, the final results, as indicated by the indicator field in Figure 2.2, shows all the employees satisfying the above conditions.

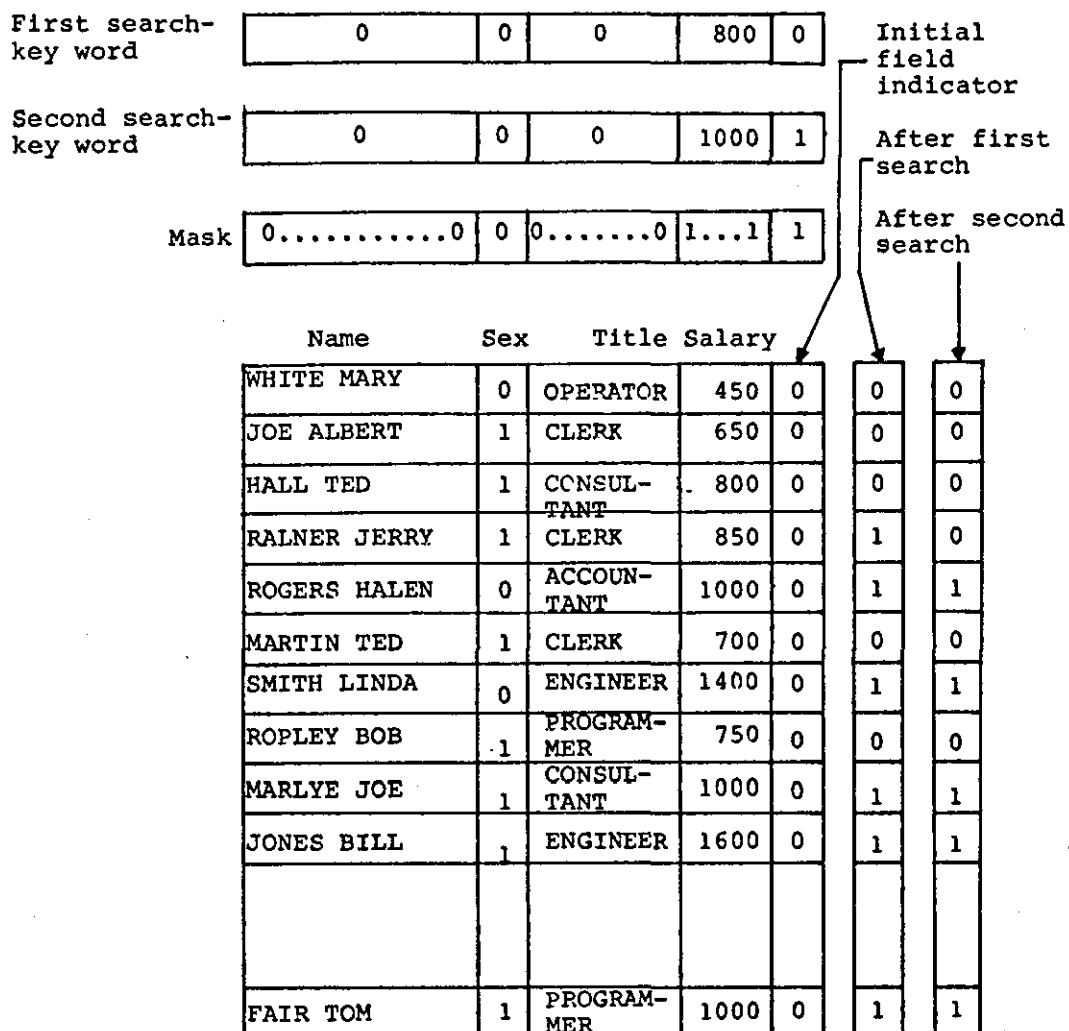


FIGURE 2.2: AN EXAMPLE OF THE OPERATION OF AN ASSOCIATIVE MEMORY

2.2.2.2 THE TAXONOMY OF THE ASSOCIATIVE PROCESSORS

From the architectural point of view, Associative Processors are classified as SIMD computers where, in addition to the associative memory addressing property, arithmetic and logic data transformation operations can be performed over many sets of arguments under a single instruction being propagated from the central control unit.

A classification of these types of processors, based on the comparison process followed by the associative memory, has identified four architectural Associative Processor categories: the fully parallel, the bit-serial, the word-serial and the block-oriented associative processors which are briefly described below.

- i) The fully-parallel class of computers, one of the two most widely known associative processors categories (the second being the bit-serial class) can be further distinguished into the word-organised and the distributive logic types. In the first sub-category, the comparison logic is based on each bit-cell of every memory word and the logical decision is available at the output of every word. Consequently, the comparison process can be performed in either a parallel-by-word or parallel-by-bit fashion, or both. From the operational point of view, these are the simplest and fastest form of fully-parallel associative processors as compared to other associative structures. On the other hand the high hardware complexity involved in the provision of a separate logic circuitry for every single bit-cell has constrained the implementation of this type of system to only experimental purposes. Figure 2.3 shows a general structure of such

computers. The main characteristic of the second sub-category of the fully-parallel computers is the association of the comparison logic function with each character-cell or each group of character-cells. The first associative processor computer of this type, the DLAP - 'Distributed Logic

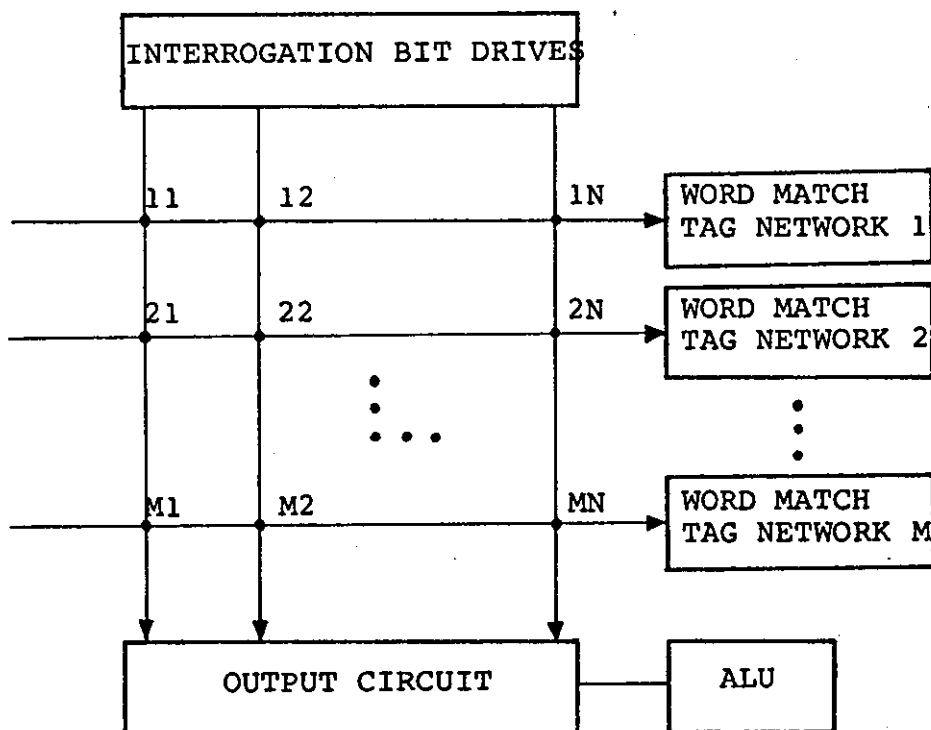
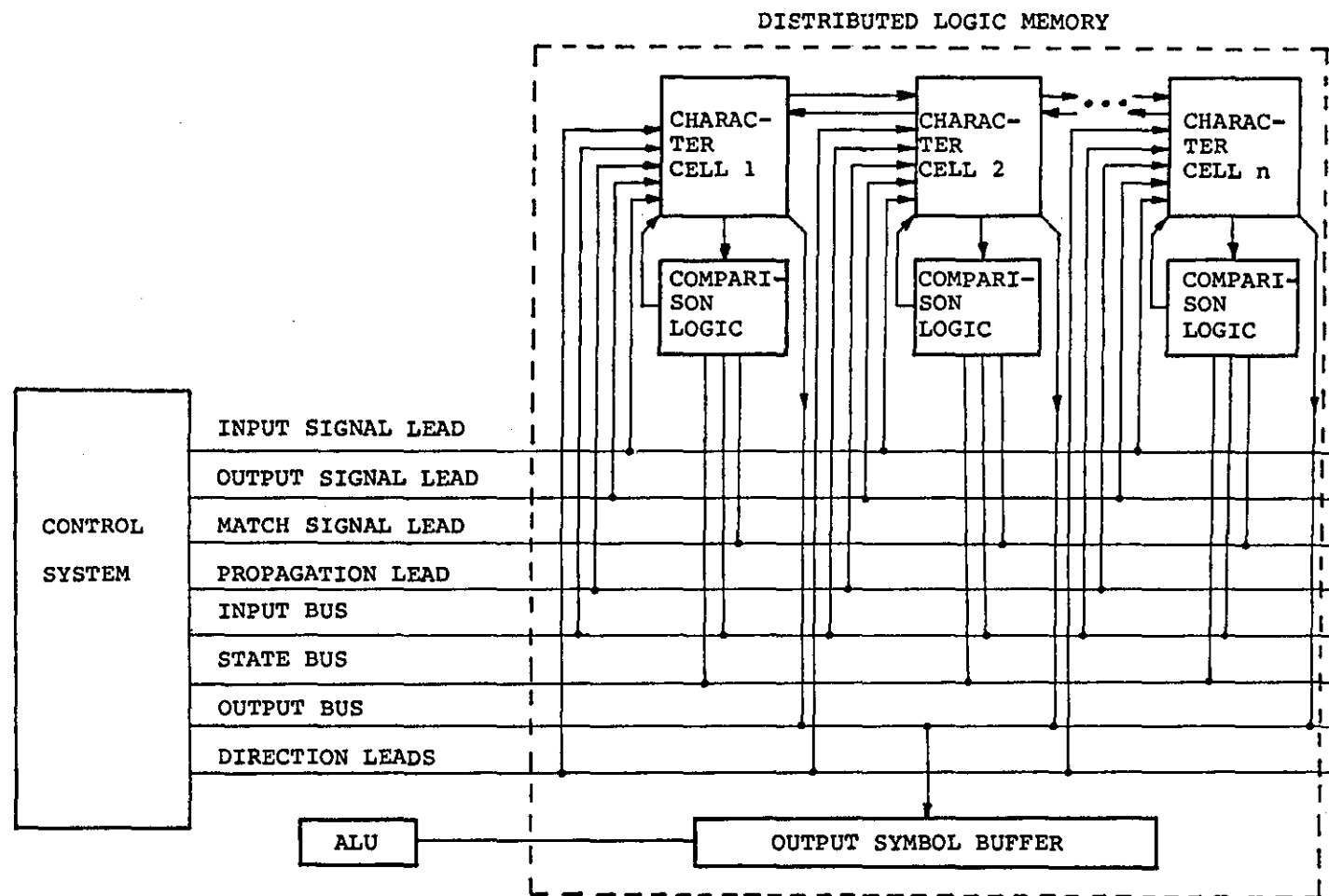


FIGURE 2.3: General Structure of a fully-parallel word-organised Associative Processor where each cross point is a bit-cell comparison

Associative Processor', was proposed by Lee in 1962 [Lee 1962] whose architectural block diagram can be seen in Figure 2.4 and the best known implemented computer of this type is the PEPE, developed by Bell Laboratories for the US Army Advanced Ballistic Missile Defence Agency. Several extensions to Lee's original system have been proposed mainly to increase the potential computational throughput by solving many of the then

FIGURE 2.4: GENERAL STRUCTURE OF THE DIAP AS PROPOSED BY LEE IN 1962



encountered problems. For example, a distributed associative memory with two cell-states instead of one for each character-cell was proposed by Lee and Paull [Lee 1963] in order to tackle many of the information retrieval problems such as cross-retrieval, erasing, gap closing and preference. Another proposal, due to Gaines and Lee [Lee 1965], suggested a redesign of the logic circuitry using two different-purpose cell-state elements, called the match flip-flop and the control flip-flop, in order to overcome skewed propagation timing. Consequently the memory was able to perform two simultaneous operations, shifting and marking strings. Crane and Githens [Crane 1965] proposed an extension of Lee's system to a two-dimensional distributed logic memory, using a large number of identical processing elements to increase the execution rate of arithmetic and logic operations.

- ii) The bit-serial Associative Processors emerged as a compromising result when attempting to tackle the economical and technical problems seen in the previous class of Associative Processors. The basic idea behind this class of computers was the use of parallel processing on Vertical data concepts developed by Shooman in 1960 [Shooman 1960].

Since then, several proposals based on the above concept have been made for Associative Processors. Kaplan [Kaplan 1963] proposed a bit-serial associative memory used as a sub-system for a general-purpose computer. A memory register is used in the data communication between the main memory and the bit-serial associative subsystem. Ewing and Davies (see [Ewing 1964]) were the first to propose the design logic of such a computer. Figure 2.5 shows the associative memory

organisation and the ALU where each intersection of a word line and a bit line represents a bit-storage. The parallel processing of a search operation takes place at bit-slices, being detected by the bit-column-select logic, through the word logic associated with each word line. The logic is identical to all words and consists of a sense amplifier, storage flip-flops, a write amplifier and a control logic. Another proposal for the implementation of a bit-serial associative memory using conventional destructive-readout memory elements was due to Chu [Chu 1965]. As a result of the two-dimensional read/write capability offered by this memory, called horizontal or vertical memory operations, two types of computer organisations are possible, namely the conventional computer organisation for the horizontal operation mode and the bit-serial computer organisation otherwise.

Bit-serial associative processors have been implemented using 2 D core search memory (see [Harding 1968] and [Stone 1968]). STARAN, one of the best well-known bit-serial associative computers, was developed by the Goodyear Aerospace Corporation. It consists of a control system, an operational number of up to 32 associative array modules, each of which is 256-word x 256-bit two-dimensional access memory, 256 simple processing elements and a selector. More significantly is the permutation or interconnection network through which the appropriate operands transfer between the processing elements and the memory modules is obtained for a full maximisation of the parallel search operation. In addition to the high speed Input/Output capabilities, the STARAN computer can be connected easily to most conventional sequential systems. Consequently, this hybrid system can increase considerably the throughput rate of many time-consuming applications such as

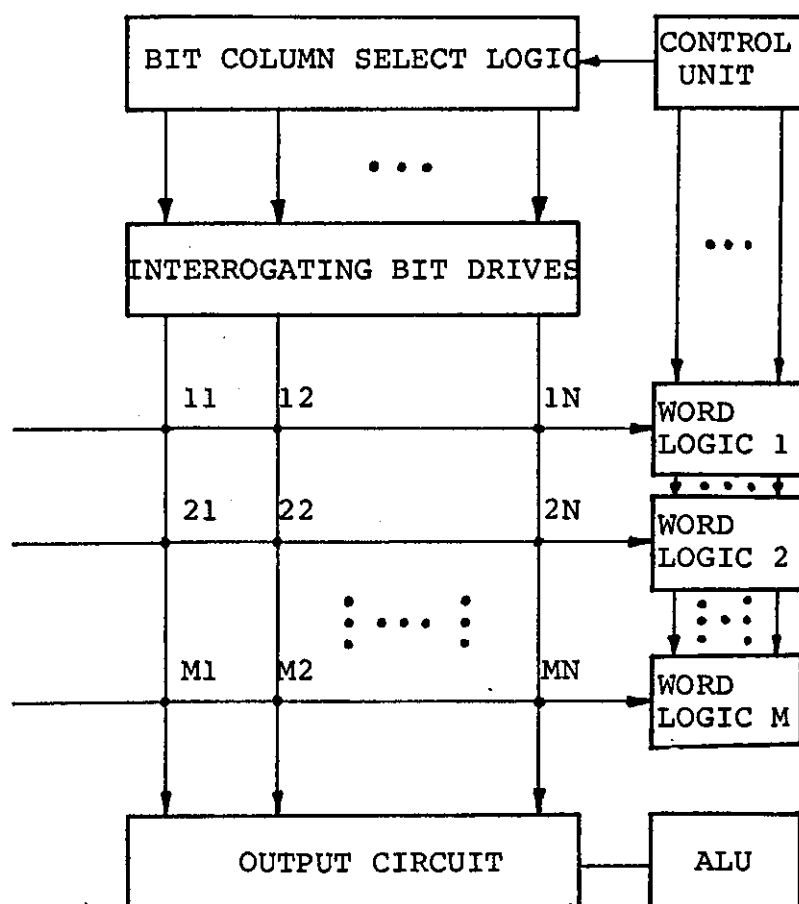


FIGURE 2.5: ASSOCIATIVE MEMORY AND ALU ORGANISATION OF A BIT-SERIAL TYPE COMPUTER WHERE EACH INTERSECTION OF A WORD LINE AND A BIT LINE REPRESENTS A BIT-STORAGE

air-traffic control, signal processing and data management systems. STARAN is, however, not the only one bit-serial associative processor, other examples include the OMEN computers, developed by Sanders Associates, the Hybrid associative processor, developed by Hughes Aircraft Co (see [Love 1973]), the RPA - 'Raytheon Associative Processor, (see [Couranz 1974]), the ALAP - 'Associative Linear Array Processor' - see [Finnila 1977]) and the ECAM - 'Extended Content Addressed Memory' - (see [Anderson 1976]).

- iii) Word-serial associative processors which essentially represent a hardware implementation of a simple search program loop have not been commercially developed due to the fact that they do not promise a high executional speed. Their main speed gain when compared to a programmed search in a standard sequential computer is achieved by a reduction in the instruction decoding time since the search operation in a bit-serial computer requires only one instruction. The first experimental model of such a computer was represented by Crojut and Sottit in 1966 [Crojut 1966]. Their model was based on a word-serial associative memory with operational characteristics very similar to that of a disc or a drum. The memory used n ultrasonic delay lines, where n is the number of bits/word, operating at 100 MHz (million Hertz) with a delay time of 10 sec. The information traffic through the delay lines is assumed by the rewrite control logic which has the capability to either recirculate the same information or input new data. Individual memory words can be interrogated or updated at the exit of the delay lines. Also a synchronising clock pulse generator, or a stable oscillator (STALO) was used to advance the address counter. Figure 2.6 shows the hardware structure

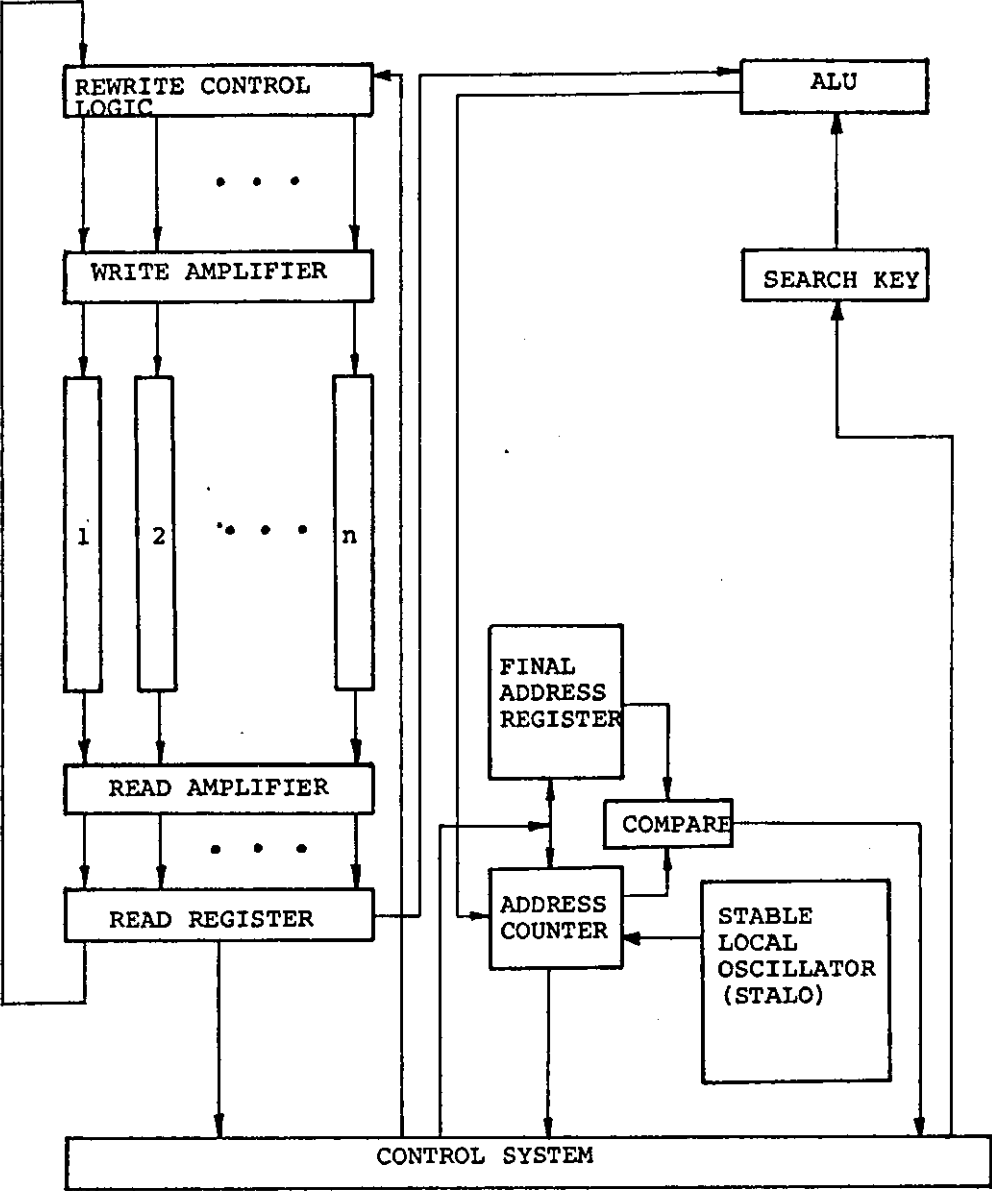


FIGURE 2.6: GENERAL STRUCTURE OF A WORD-SERIAL ASSOCIATIVE PROCESSOR COMPUTER

of such a computer.

- iv) Finally, the block-oriented Associative Processors which can be seen as a compromise between the low speed word-serial and high-cost associative processors, are based on a rather large rotating storage with limited associative capabilities. This type of computer provides low-cost processing and is particularly suitable for applications with a large data storage requirement. Several models of this type of computer such as the RAPID - 'Rotating Associative Processor for Information Dissemination' - Computer, presented by Barhami in 1972 [Barhami 1972] have been developed. The design of the RAPID, whose general structure is outlined in Figure 2.7 was based on Slotnick's and Parker's concept of using logic-per-track devices (i.e. disc memories with a head and some logic associated with every track) and Lei's distributed logic memories for information storage and retrieval applications. The high rate of data transfer between the head-per-track disc and the associative memory makes the RAPID computer suitable for problems requiring large storage.

Recapitulating, Associative Processors which form a sub-class of the general SIMD computers, are mostly characterised by the efficient search capabilities offered by their associative memories. A classification of these processors, based on the associative memory organisation, has identified four distinct categories: the fully-parallel, the bit-serial, the word-serial and the block-oriented associative processors; the first two being the most widely known type of computer.

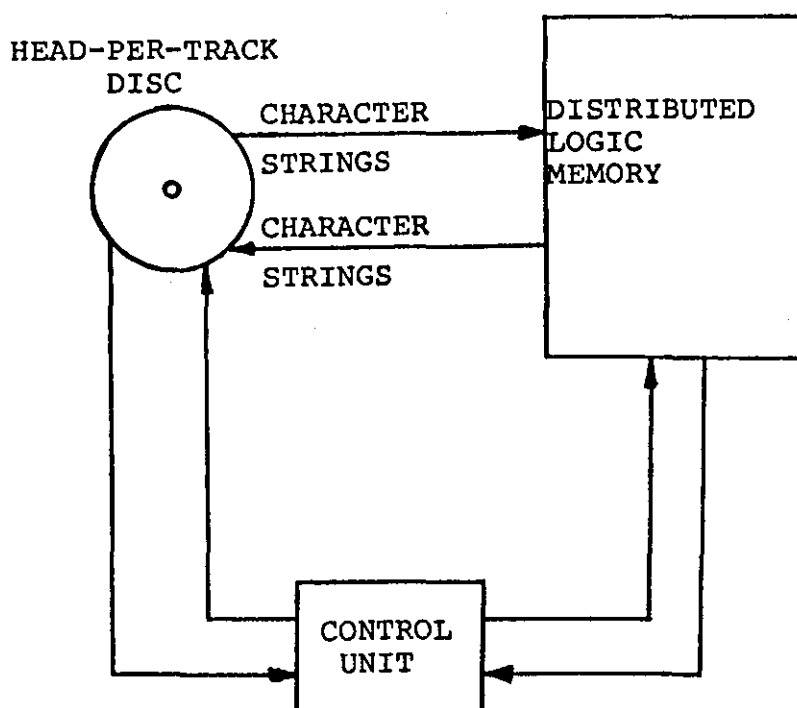


FIGURE 2.7: BLOCK-ORIENTED ASSOCIATIVE MEMORY STRUCTURE IN THE RAPID COMPUTER

The early designed associative processors offered limited associative processing at a high implementation cost. With the advent of LSI technology and the use of new architectural design concepts, these types of computers became gradually more practical. PEPE and STARAN are the most widely known fully-parallel and bit-serial computers respectively.

Already, various associative processors with a storage capacity of several hundred million bits have been built. Also many experimental models have been proposed for later implementation whenever technology development allows it.

2.2.3 ARRAY PROCESSORS

The early interest in the Parallel Processor area initially appeared in the investigation of machines that were arrays of processors connected in a four-nearest-neighbour fashion, such as the Von Neumann's Cellular Automate (see [Von Neumann 1968]) and the Holland Machine (see [Holland 1959]). Eventually, as a result of the growing interest in this form of computer, Parallel Processors with a central control mechanism that controlled the entire array and operating in a SIMD manner began to emerge.

The description of the main characteristics of the Parallel or Array processors shall draw heavily on the ILLIAC IV (see [Barnes 1968]) and the ICL DAP (see [Reddaway 1973]) systems. This is mostly because of the profound effect that the former computer has had on this class of systems and the ease to access the latter machine from Loughborough University through the Janet Network. Finally, a brief description of the SOLOMON computer series (which preceded the design of the ILLIAC IV is also included.

All the systems in the Array Processor class can be identified by their major components, structured in a number of various and different ways:

1. a number of identical Processor Elements (PE's) synchronously operating on different data streams proliferated from
2. a number of memory banks, not necessarily equal to the number of the PE's through
3. a communication network with
4. some form of local control and finally
5. some form of global control.

Two simplified array computers are shown in Figure 2.8.

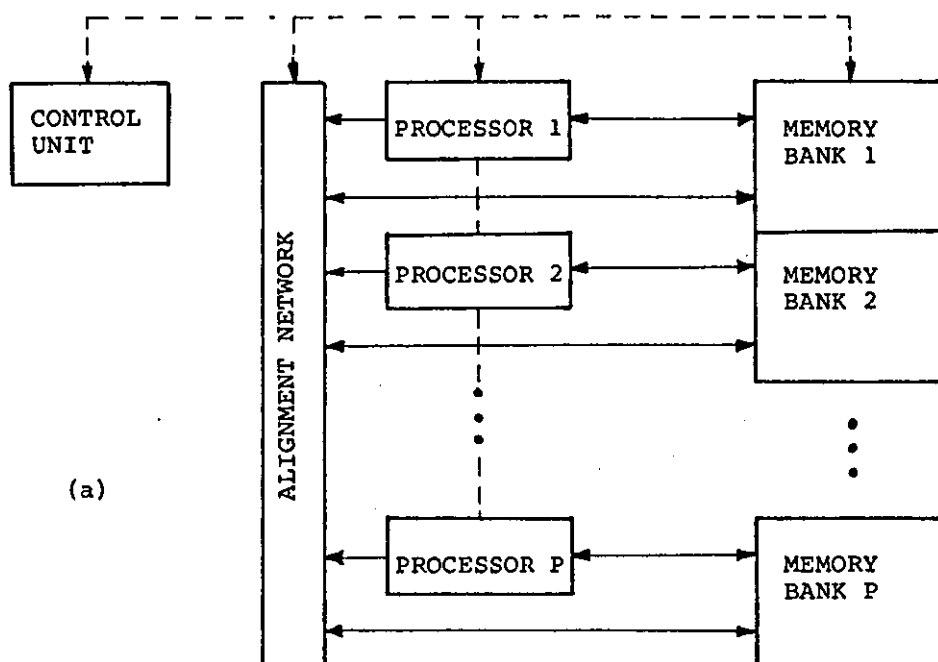
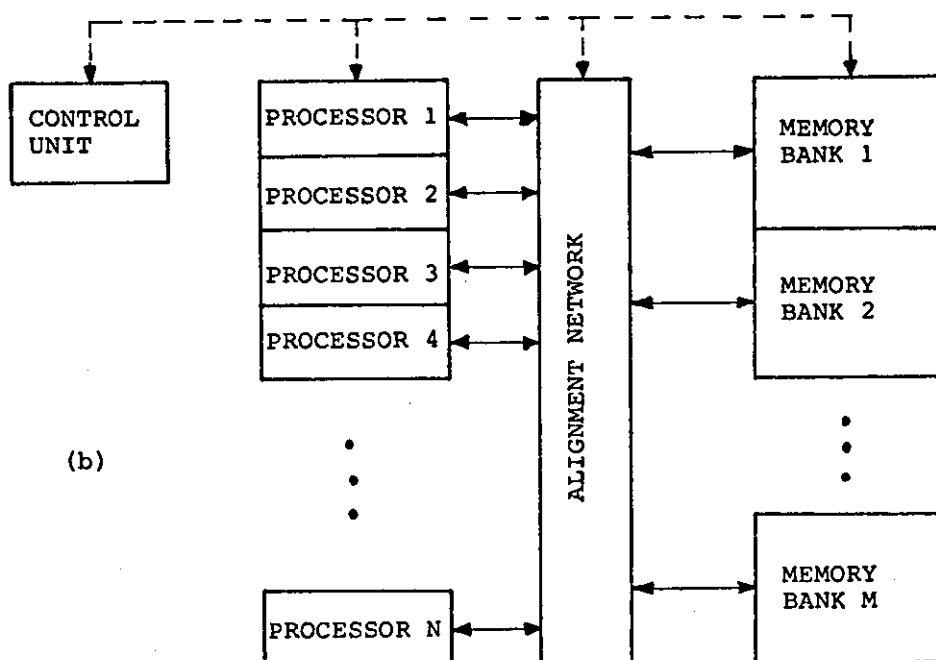


FIGURE 2.8: A GENERAL SIMD ARCHITECTURE ALLOWING EITHER AN IDENTICAL NUMBER OF PROCESSORS OR MEMORIES (a) OR A DIFFERENT NUMBER OF PROCESSORS AND MEMORIES TO BE CONNECTED (b)



The control unit which is usually a computer itself with its own arithmetic and logic unit, memory and registers, differs from the other processors in that it can execute scalar and control instructions (including conditional branch instructions). The processor elements which lack this ability since they must all be kept in synchronisation, do not generate their own instructions, but they all receive the same sequence of vector instruction from the control unit. A local on-off control unit is used to permit processors to either execute or ignore certain broadcasted vector instructions.

An array computer with p processors can produce a speed-up of p with respect to a uniprocessor system. However there are several degradation factors which reduce the actual speed-up. Specifically, these factors are:

1. Sometimes the execution of a vector of elements whose length is not a multiple of the number of processors does not use all the processing units, thus reducing the actual throughput;
2. Typical algorithms contain scalar instructions that cannot be overlapped with vector instructions. This keeps the array of processors idle while a scalar instruction is executed;
3. Several memory conflicts arise when accessing more than one vector element from the same memory bank. In order to deal with these conflicts, several storage techniques which will be shortly reviewed below, for vectors and matrices have been proposed to reduce this degradation;
4. Finally, limitations in the interconnection network incur some delay in the data transfer which has to be performed simultaneously from the memory modules to the corresponding processors.

Due to these degradation factors, it is very difficult to predict the performance of a specific application without making a detailed analysis of the problem. Besides the ILLIAC IV and ICL DAP computers, examples such as the BSP - 'Burroughs Scientific Processor' (see [Kuck 1982]), and the MPP - 'Massively Parallel Processor' - (see [Batcher 1979]), developed by the Goodyear Aerospace Corporation, belong to the array processor type of computers.

2.2.3.1 INTERLEAVED PARALLEL MEMORIES

Processors utilisation was also improved by the fact that the time spent in accessing the memory was greatly reduced. The standard and more significant way to do this is to increase the bandwidth of the memory by dividing it into several modules, or Parallel Memories that can be accessed simultaneously. More specifically, if in an m -parallel memory system, the successive addresses are assigned across the memory banks, module m , it will be called an interleaved parallel memory system. Other ways of reducing the access time are to add a cache memory and to include fast registers in the processors. The use of these techniques is related to the particular system organisation.

Several techniques have been proposed to solve the problem of conflicts that may arise in the use of these types of memories. Since most serious difficulties generally occur in two-dimensional problems, a typical application is the matrix computation, a simple example of storing a two-dimensional array $A(m \times n)$, where $m=n=4$, is considered. Figure 2.9(a) shows A stored in a format known as **straight storage** where each one of the 4 memories contains a column of the matrix. This is suitable for row or diagonal access but

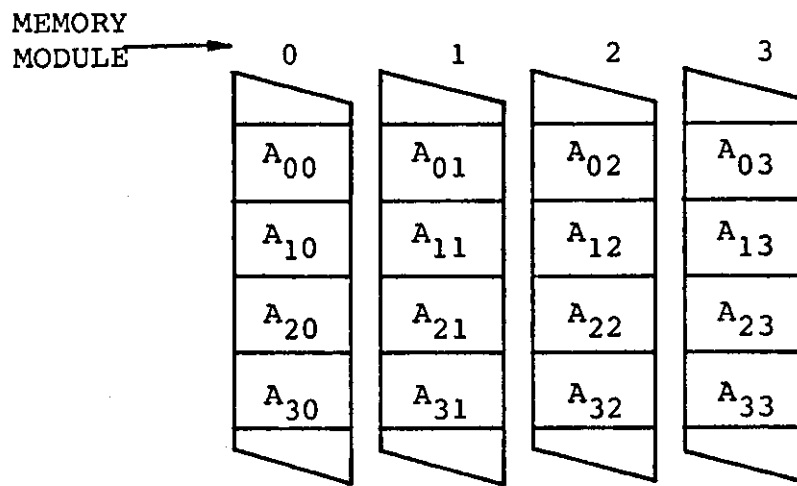


FIGURE 2.9(a): STRAIGHT STORAGE FORMAT OF A 4x4 MATRIX IN A FOUR-MEMORY SYSTEM

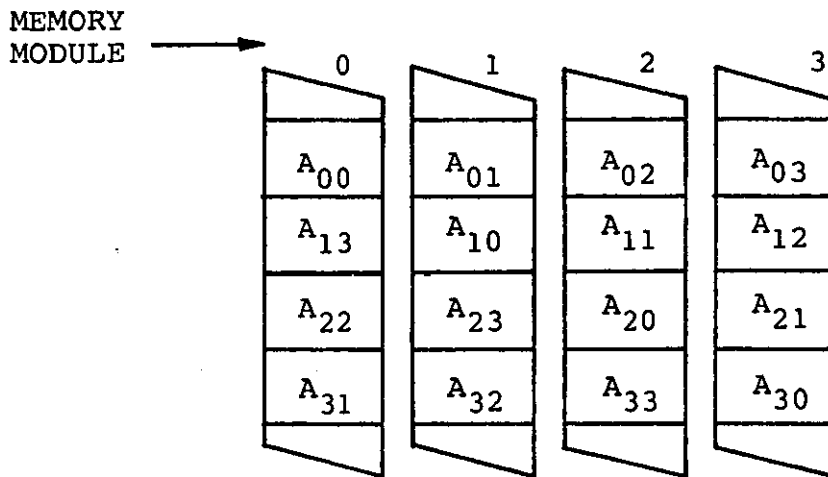


FIGURE 2.9(b): SKEWED STORAGE FORMAT OF A 4x4 MATRIX IN A FOUR-MEMORY SYSTEM

presents some conflicting problems when accessing a column since 4 cycles are required to fetch a single column. However by skewing the data across the memory banks as shown in Figure 2.9(b), elements of a row or a column can be accessed during one single memory cycle but not the diagonals.

Skewed storage is a relatively inexpensive way to enhance the processing capability of an array processor system. It requires each processor to have a private index register, and it requires cyclic interconnection. Additional cost in processing time is involved when using skewed storage instead of straight storage. Thus if an algorithm requires access to rows only straight storage is slightly preferable to skewed storage. On the other hand, if both rows and columns need to be accessed as vectors, then skewed storage is strongly preferable to straight storage. If access to columns is the only requirement, then the matrix should be stored in transposed form in the straight storage format.

Other mechanisms of manipulating parallel memories can be seen in the TI ASC and the STAR 100 where the use of the physical array transposition technique offers the capability of accessing data from the rows, columns and diagonals but at a high transposition time overhead.

2.3.3.2 THE INTERCONNECTION NETWORKS

One of the most currently active research areas in computer architecture is the interconnection networks since they represent the accumulation of a large number of design decisions made before the implementation of the actual architecture. These systems range in organisation from two processors sharing a common memory

(Multiprocessor) to a large number of relatively independent computers connected over geographically long distances (distributed computing). Anderson and Jensen presented a naming scheme, or taxonomy, which was strongly biased towards distributed computing systems [Anderson 1975], explicitly avoiding SIMD machines such as the ILLIAC IV and PEPE. However we shall introduce this subject, while slanting the applications towards processor array interconnections.

The interconnection networks can be generally distinguished into two types, the bus and the alignment networks with a basic difference between them: while the former allows only a single one-to-one communication to take place at any given time, the latter allows several one-to-one (parallel data and control transfer) or one-to-many (allowing one unit to broadcast to many units in parallel) communications. It follows that the bus network is less expensive but a slower network than the other.

Furthermore, the alignment networks can be topographically sub-categorised into static and dynamic networks. A static network is characterised by the required dimensions for layout. Examples range from one-dimensional structures to hypercube networks. In Figure 2.10, we can see examples of one, two and three-dimensional networks. On the other hand, the dynamic networks are distinguished into the single-stage, multiple-stage and crossbar types of networks. The single-stage network consists of a single stage of switches. The nearest neighbour network and the perfect shuffle networks are examples of this type of network (see Figure 2.11). A more generalised connection network, where every input is connected to every output channel through a crosspoint is the crossbar switch. Figure 2.12 shows two representations of the crossbar switch from

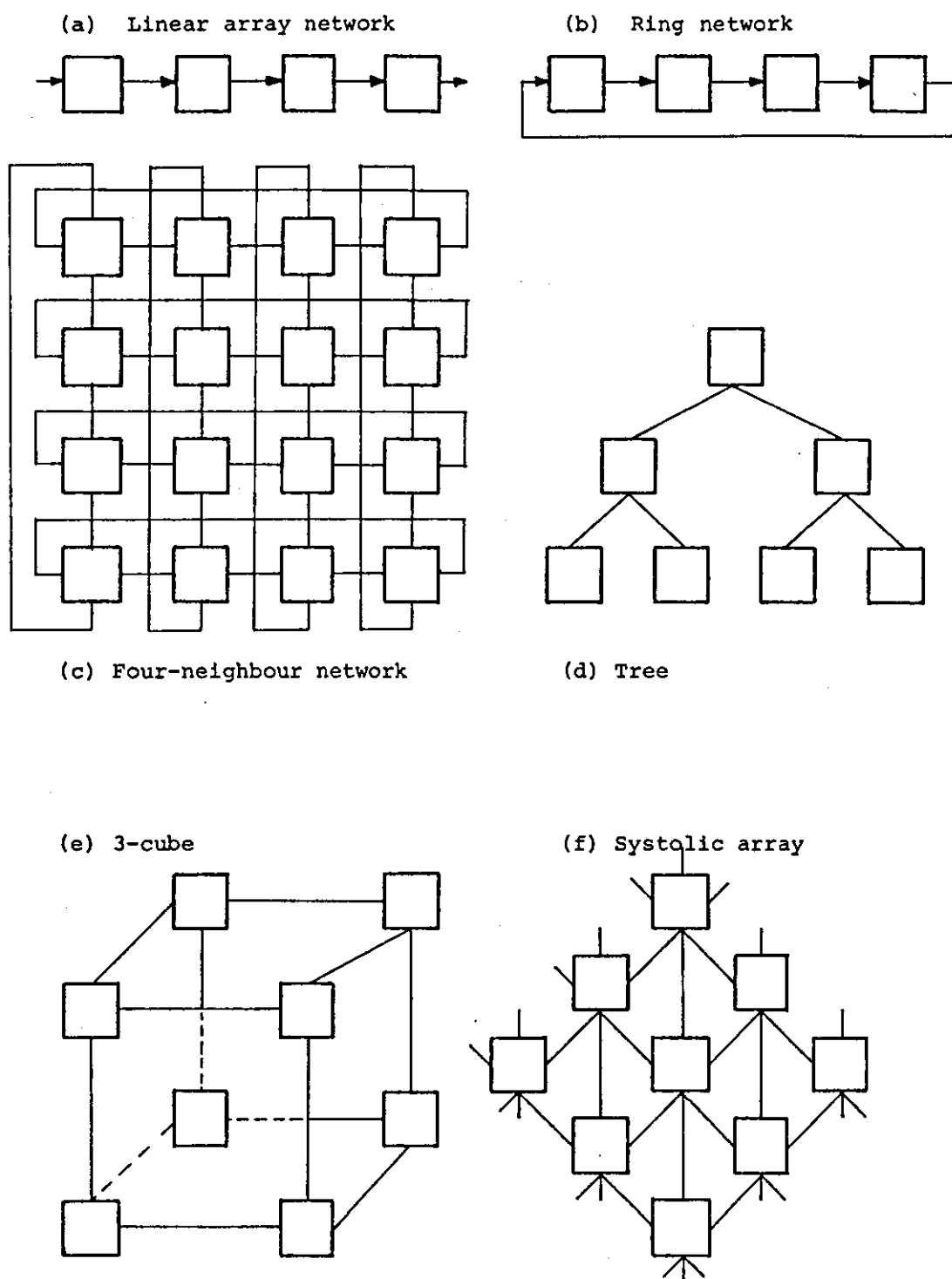


FIGURE 2.10: EXAMPLE OF 1, 2 AND 3-DIMENSIONAL INTERCONNECTION SYSTEMS

four inputs to four outputs. Finally, the multi-stage networks which can provide a cheaper alternative to the complete connection as offered by the crossbar switches are based upon a number of interconnected 2x2 crossbar networks organised into several stages. In Figure 2.13 we can see two multi-stage networks, the binary Bene's and the indirect binary n-cube networks.

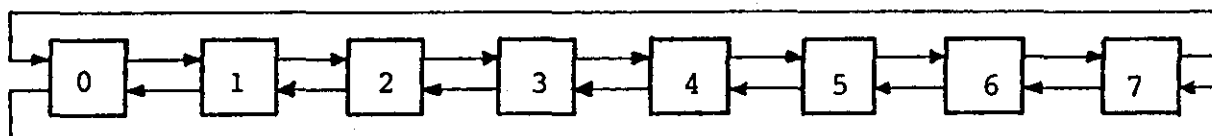


FIGURE 2.11(a): THE NEAREST-NEIGHBOUR NETWORK

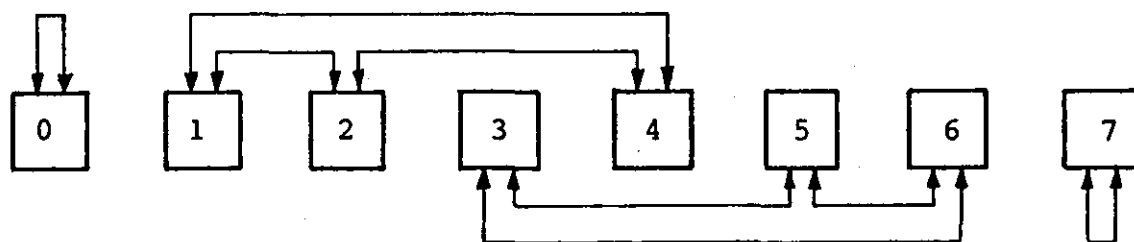


FIGURE 2.11(b): PERFECT-SHUFFLE NETWORK

2.2.3.3 IMPLEMENTED ARRAY PROCESSOR COMPUTERS

In this concluding paragraph about the array processor computers, we shall briefly describe the main architectural characteristics of the two most significant computers, the Burroughs ILLIAC IV and the ICL DAP computers. This choice was mainly motivated by the fact that the former has had a great impact on the parallel processing concept and the latter can be accessed from Loughborough University. However,

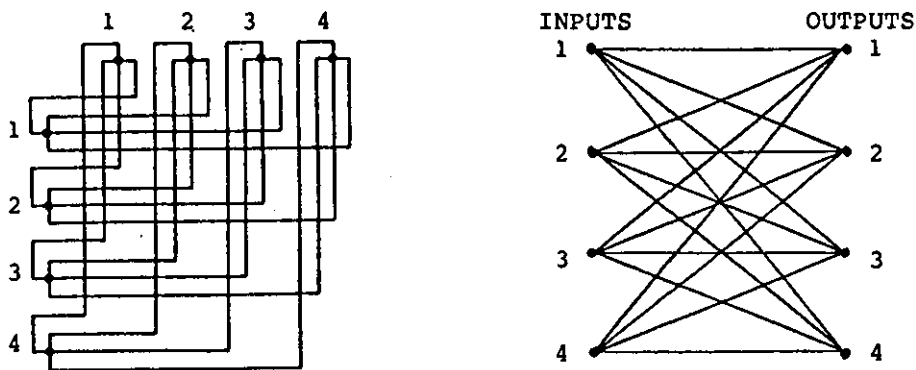


FIGURE 2.12: TWO REPRESENTATIONS OF THE CROSSBAR SWITCH FROM FOUR INPUTS TO FOUR OUTPUTS

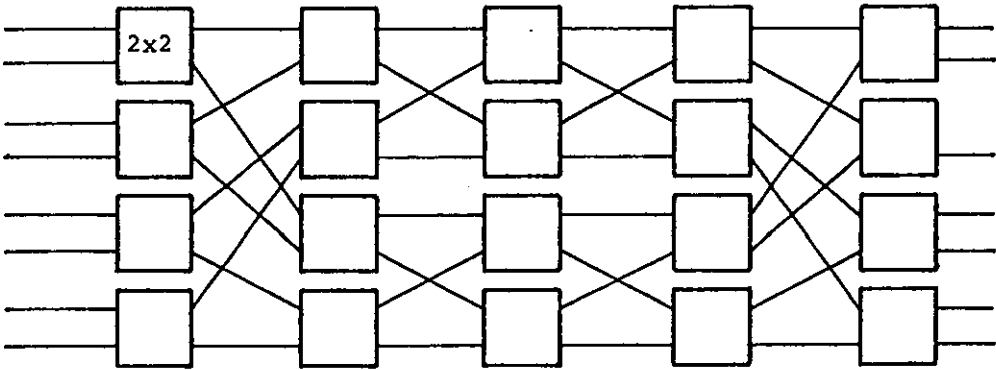


FIGURE 2.13(a): THE BINARY BENES NETWORK USING 2x2 CROSSBAR SWITCHES

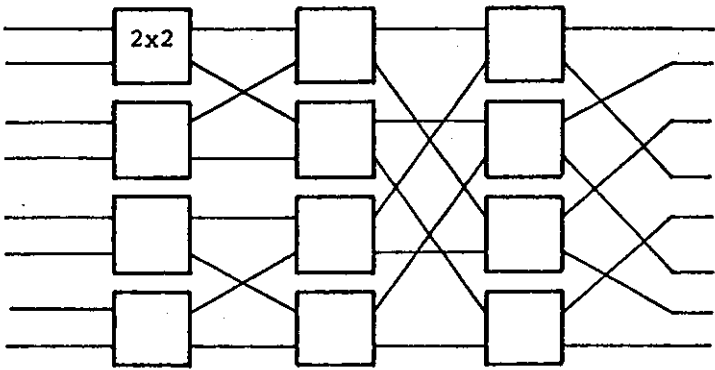


FIGURE 2.13(b): THE INDIRECT BINARY_n-CUBE NETWORK

no attempt was made to compare the performance of this system since it was outside the scope of this current research.

The design of the ILLIAC IV computer by Slotnick et al in 1966 was strongly based on two previously designed but never built array processors, the SOLOMON I and the SOLOMON II computers. These machines, each of which contains an array of processing elements with four-nearest-neighbour connections, were primarily designed to solve many problems involving differential equations, matrix manipulations, weather data processing, and linear algebra. SOLOMON I was a bit-serial processor and every PE contained a serial accumulator. The serial arithmetic concept, although quite flexible, was found to be far too slow for the intended applications and consequently, the SOLOMON II arithmetic units were switched to 24-bit floating point units.

The eventual development of the SOLOMON computers led to the ILLIAC IV where the arithmetic units were improved further to a 32-bit word length. Also, the array configurations changed from four 8x16 quadrants to four 8x8 PE quadrants. However only 1/4 of the original designed configuration was actually built by Burroughs and delivered to NASA Ames Research Centre, California, in 1972.

The Control Unit (CU) of the ILLIAC IV as diagrammed in Figure 2.14 consists of five major components: ILA - 'Instruction Look Ahead', ADVAST - 'ADVanced STation', FINST - 'FINal STation', MSU - 'Memory Service Unit' and TMU - 'Test and Management Unit'. Control instructions are fetched from the PE memories, which form an integral part of the physical memory, and purged into the ILA. Scalar instructions are examined and executed by the ADVAST subsection whereas vector instructions are decoded by the FINST

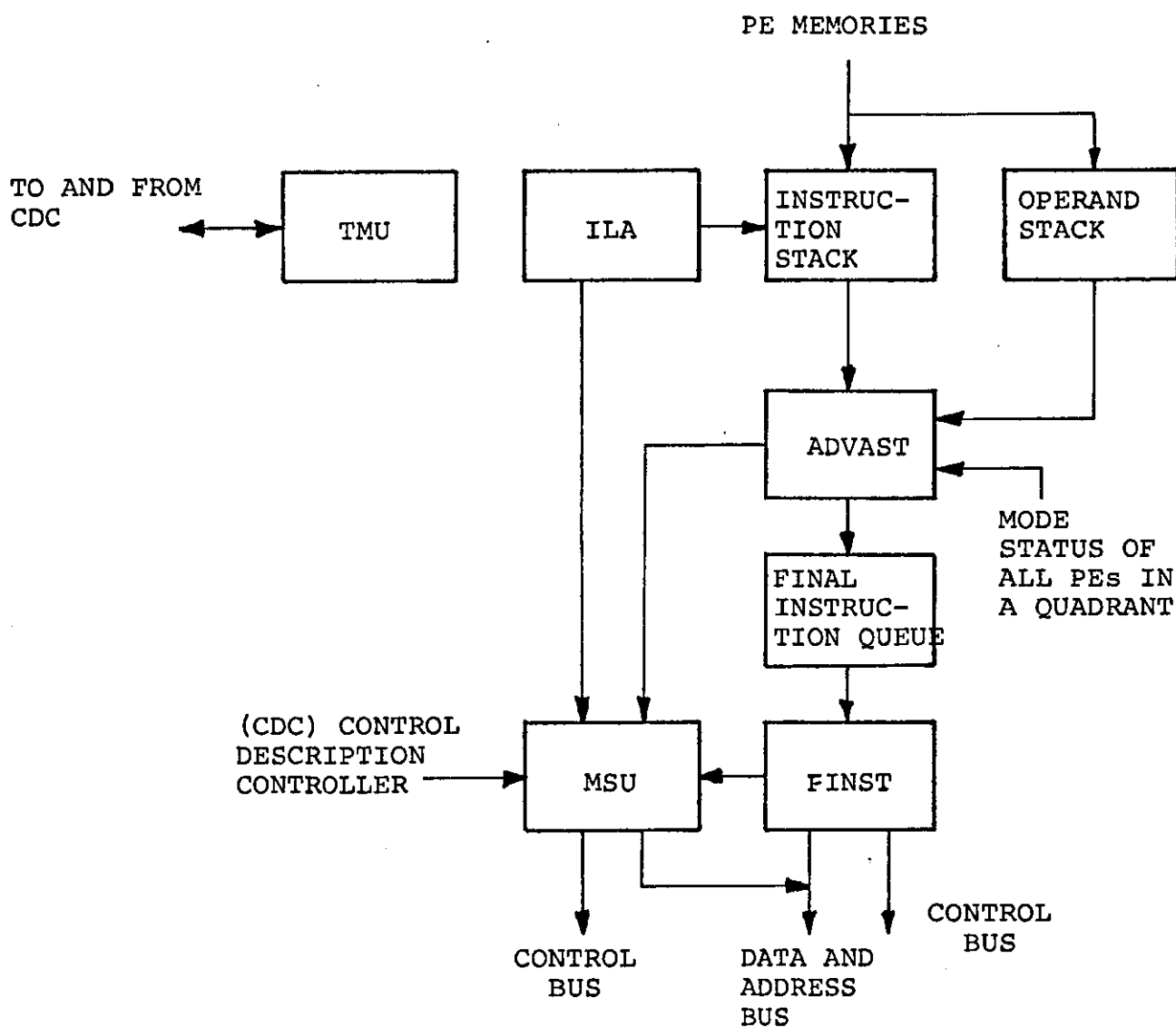


FIGURE 2.14: FIVE MAJOR COMPONENTS OF THE ILLIAC IV CONTROL UNIT

before transmitting them to the PE for execution. The CU, also contains four general-purpose accumulators, several control registers, a 64-bit scratch pad and quadrant control registers.

The processing unit consists of the Processing Element, its memory and the MLU - 'Memory Logic Unit'. The Processing Element memory is built out of 2048 words of 64-bit thin-film memories with an access time of 240 ns. The total 128 Kwords physical memory is also backed by a large disc as a secondary storage. The maximum processing rate achieved was approximately 50 Mflops/s which is almost one quarter of the expected performance.

Several high level languages that could exploit the systems' parallelism have been proposed for the ILLIAC IV: the Algol-like TRANQUIL (see [Abel 1969]), the Pascal-like ACTUS (see [Perrott 1978]), the GLYPNIR (see [Lawrie 1975]) and the CFD FORTRAN (see [Stevens 1975]).

In conclusion, the ILLIAC IV was regarded as a failure, not only from the high cost, since it used the very expensive state-of-the-art-plus technologies, but also from several major bottlenecks (see [Hockney 1977]) which were identified by the Burroughs contractor. Based on the experiences gained from the ILLIAC IV development, Burroughs built the Burroughs BSP, which although similar to its predecessor, is designed to circumvent many of the problems encountered in the ILLIAC IV (see [Jensen 1978]).

Unlike the ILLIAC IV and many other supercomputers, which relied on the state-of-the-art-plus switching technologies, the pilot model of the ICL Distributed Array Processor (DAP) was originally built from relatively modest technology and at fairly low levels of

integration, thus providing a relatively cheap product capable of a very wide performance range depending on the application (see [Reddaway 1977]). Conceptually, the design of the pilot DAP was similar to that of the initial SOLOMON computer and consisted of a two-dimensional (32x32) array of one-bit slave processors.

However, the design of the DAP introduced two new contributions to the SOLOMON concept: the first one was characterised by the hardware feature which effectively slices the array in two orthogonal directions. Either direction of the array can be aligned to a set of registers of the 'Master Control Unit' - (MCU) using a separate orthogonal data highway which threaded the rows and columns of the PEs. These highways which served the purpose of collecting and broadcasting data to slices of the DAP array, was the most significant element in providing the DAP with much of its flexibility in manipulating data. The second contribution relied on the manner in which DAP was integrated into a complete system. Not only did it emulate the memory of an ICL mainframe computer to which it was attached, but also it was capable of processing data autonomously in a highly parallel manner.

One of the first three implemented computers of this type was delivered and installed at Queen Mary College (UK) in 1980, consisting of a (64x64) matrix of PEs arranged in the same geometry and each having 4 Kbits of memory. This gives a total of 2 Mbytes of memory for the attached top-end ICL 2900 mainframe computer. In Figure 2.15 we can see the major subsections of the DAP computer.

Another feature of the DAP's design which helped to avoid the Von Neumann bottlenecks is the inclusion of the PE logic and its associated memory on the same circuit board. Furthermore, since the

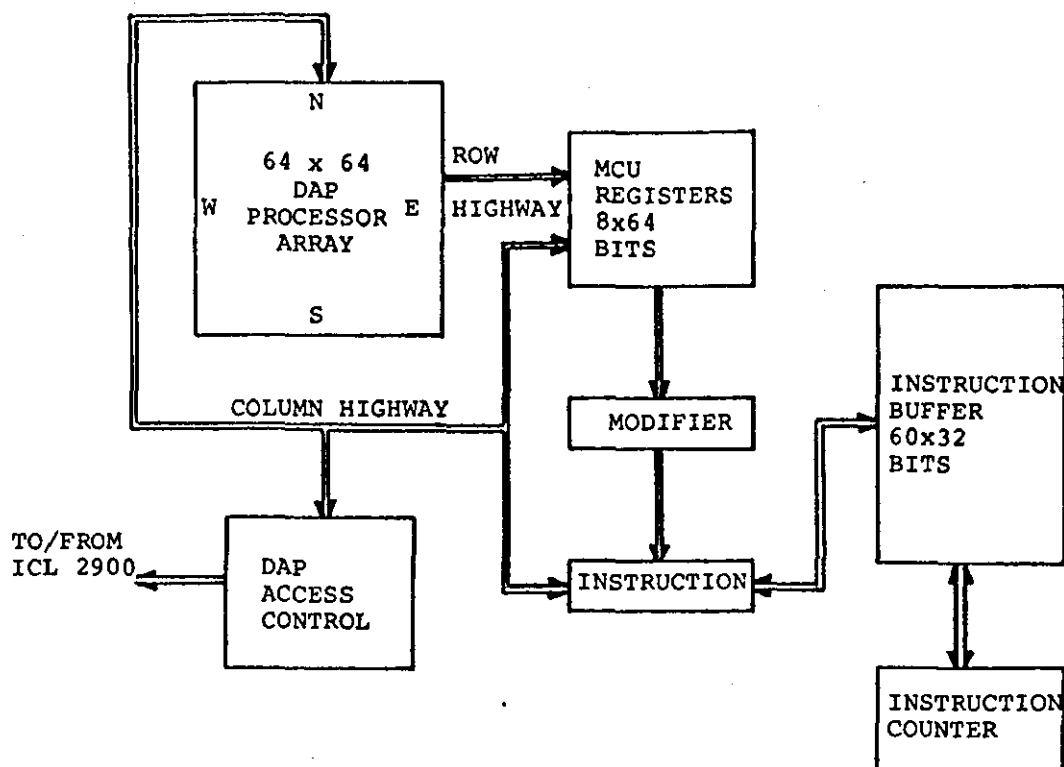


FIGURE 2.15: MAJOR SUBSECTIONS OF THE ICL DAP COMPUTER

structure of the DAP array is highly regular, the use of VLSI technology will undoubtedly increase the number of processors and memories that could be mapped on to the same chip leading to even larger arrays of PE's, e.g. a (256x256) DAP.

Every PE has three one-bit registers, A, Q, C two multiplexers and a one-bit full adder to perform arithmetic operations. The A register provides programmable control over the PE's actions, the A is an accumulator and the C register is a carry store. The adder adds Q, C and the input to the PE, giving sum and carry outputs, which are stored in the Q and C registers respectively.

Finally, the architectural description of the DAP computer is concluded with a note on a parallel FORTRAN-based language, called DAP FORTRAN. This was specially developed to take the full potential advantage of the machine's high processing power.

2.3 MIMD MULTIPROCESSOR COMPUTERS

For many years the MIMD multiprocessing structures were misunderstood and they were even mistaken for many other parallel systems, such as the array processors and the multiple-computer systems, which are less promising to achieve the high performance goals as set by the fast developing computer demands. Even though, the definition of a multiprocessor system as "a computer employing two or more processing units under integrated control", proposed by the ANSI - 'The American National Standards Institute', was insufficient since the two most significant concepts for this type of computer, i.e. the sharing and interaction concepts were not included.

The most commonly accepted definition for a true multiprocessor system was suggested by Enslow in 1977 [Enslow 1977] who, in addition to the above ANSI definition, included the two following conditions:

1. all the processors which have almost equal capabilities must share access to a common memory, I/O channels, control units and peripheral devices;
2. the entire complex is controlled by a single operating system providing the interaction between processors and their programs at the job, task, instruction and data levels.

Because of the inherent flexibility of the MIMD computers the range of applications of this type of system is generally much wider than that of the SIMD computers. Although the implementation of any application suitable for parallel processing on a MIMD system is somehow a straightforward process, however a careful synchronisation

of the allocated tasks to the processors must be undertaken. By contrast, in the SIMD computer systems, the synchronisation is performed automatically whereas the additional task allocation problem of the MIMD systems does not exist since all the processors perform the same task.

Theoretically, a p -multiprocessor (a system with p processors) system is capable of achieving a speed-up of p , however several degradation factors which are discussed later tend to make the actual speed-up smaller. Such factors can be assumed in the overheads incurred by the synchronisation mechanism, the task allocation and the shared memory conflicts, all of which are problem-oriented.

The following paragraphs will focus mainly on some significant hardware and software characteristics of the MIMD multiprocessor systems that are required to support concurrency at the lowest possible overheads.

2.3.1 MIMD HARDWARE ORGANISATION

The major motivation of the MIMD computers design is the increase in the computational speed-up by the concurrent execution of instructions, organised in several sequential streams with infrequent dependencies among them, by a large pool of processors with approximately similar capabilities. Of importance to this type of structure is the mechanism to synchronise and communicate between processors. Specially, the used mechanisms can be classified into two classes, those that use a shared memory, and those that use passing messages (see [Baer 1976]. [Enslow 1977] and [Stone 1980]). The use of the shared memory which might be a multiported main

memory, cache memory or a multiported disk, results in a faster mechanism but requires all the processors to access the shared memory. Consequently, this limits the total number of processors that the system can effectively handle. On the other hand, the mechanism based on messages has a large overhead so that it is only useful when synchronisation and communication are very infrequent [Gehrig 1982].

The general class of MIMD computers was distinguished into two main classes, the tightly-coupled and the loosely-coupled systems depending on the amount of interactions between the processing elements (see [Hayes 1978]). In the case of tightly-coupled processors, as shown in Figure 2.16, (i.e. a large number of processors sharing a common parallel memory via a high-speed multiplexed bus), the processors operate under the strict control of the bus assignment scheme which is implemented in hardware at the bus/processor interface. On the other hand, in a system with loosely-coupled processors the communication and interaction takes place on the basis of information exchange. Figure 2.17 shows a general architecture of a loosely coupled system where each processor has its own local memory. Comparing the two above classes of multiprocessor systems, the main difference lies in the organisation of the memory and the bandwidth of the interconnection network.

Several interconnection networks with different characteristics such as bandwidth, delay and cost, ranging from the shared common bus to the crossbar switch, have been proposed. However Enslow identified three fundamentally different organisations, namely the time-shared common bus, the multiport memory and the crossbar switch.

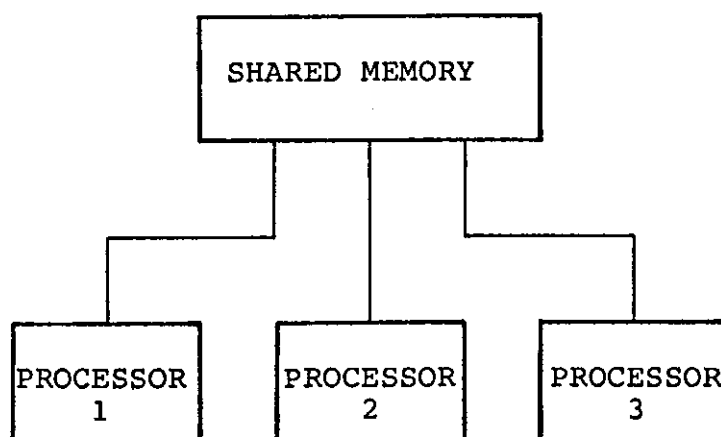


FIGURE 2.16: TIGHTLY-COUPLED MULTIPROCESSOR SYSTEM

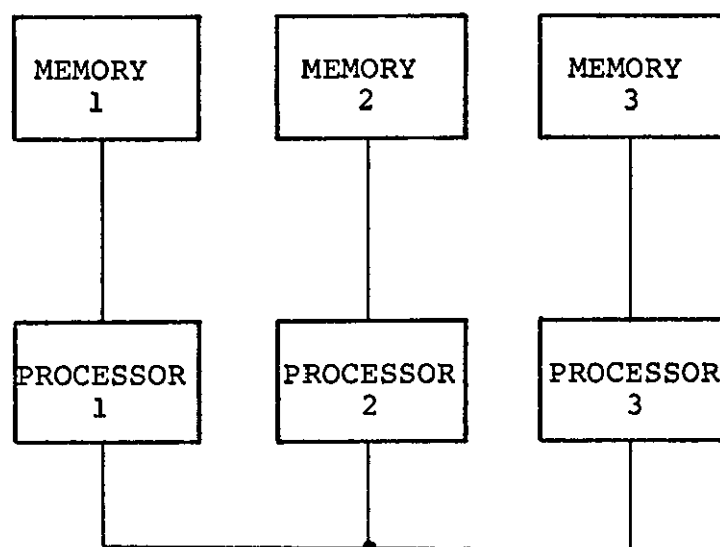


FIGURE 2.17: LOOSELY-COUPLED MULTIPROCESSOR SYSTEM

The ~~time-shared common bus~~ interconnection scheme, as illustrated in Figure 2.18, represents the simplest form of connecting all the functional units using a single bus which incorporates some arbitration logic associated with every bus/unit interface to resolve the bus request contention since only one transfer can take place at any given time. Thus, the unit wishing to initiate a transfer, a processor or an I/O unit, must first determine the availability state of the bus, then address the receiving unit as well as determining its availability and capability to receive the transfer.

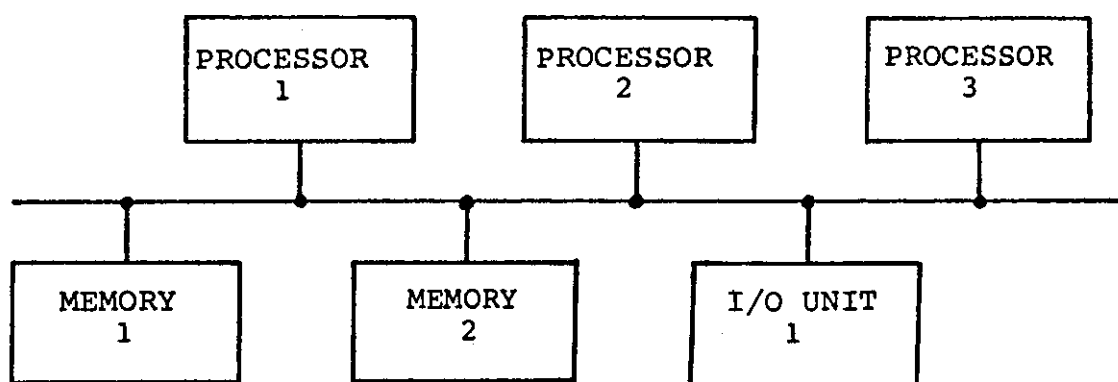


FIGURE 2.18: THE TIME-SHARED COMMON BUS INTERCONNECTION SYSTEM

By its nature, such a system is quite reliable and its cost is relatively low, however several limitations are introduced that can have serious damaging effects on both the system since a malfunction of any unit interface causes a system failure, and the total overall transfer rate. Several interconnection systems such as the use of two one-way paths and multiple two-way buses have been provided in an attempt to solve this problem of a single transfer. The former example which does not increase system complexity or diminish reliability has a comparable performance with its predecessor since a single transfer requires the use of both paths. On the other

hand, with the latter technique multiple simultaneous transfers are possible but at additional system complexity.

The most extensive and expensive interconnection network providing a separate path for every processor, memory module and I/O unit is the **crossbar switch** (see Figure 2.19). In the case that the multiprocessor system contains p processors and m memories, the crossbar requires $p \times m$ switches, each of which is capable of switching parallel transfers and arbitrating conflicting requests. In this system, the bus-interface logic required by the functional units is kept at the lowest level since some of the functions, i.e. transfer recognition and conflicts resolution, which are performed at every bus-unit interface, are assumed by the switch matrix. Consequently, such an interconnection is very complex (exponential growth for large p and m), expensive and physically large. However, the important characteristics of this system which is shown below, are the extreme simplicity of the switch-to-functional unit interfaces and the ability to support concurrent transfers for all memory modules.

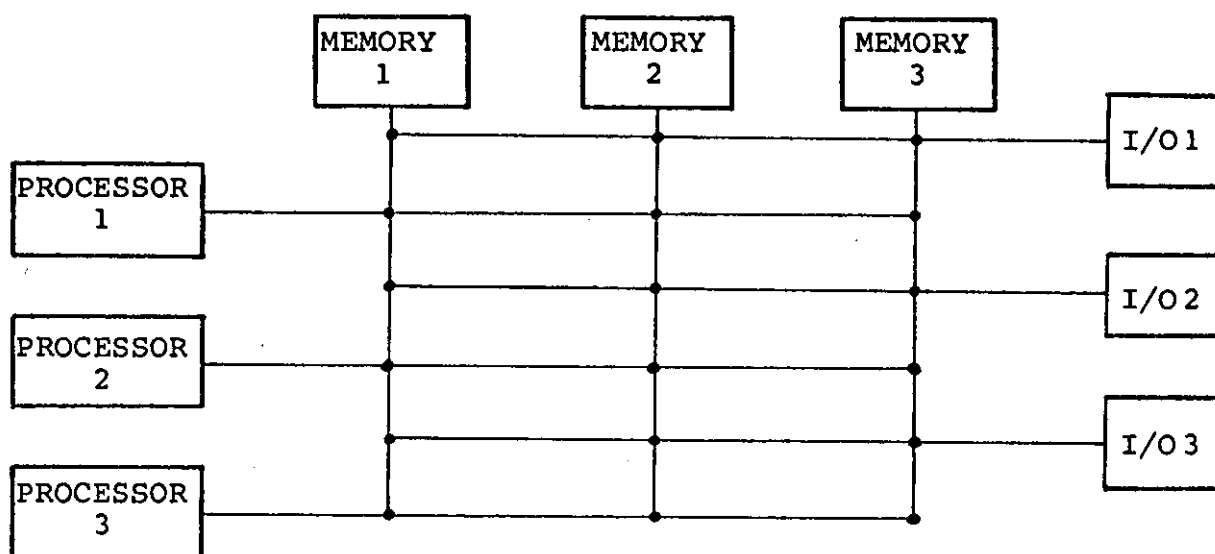


FIGURE 2.19: THE CROSSBAR SWITCH SYSTEM

The concentration of the control, switching and priority arbitration logic, which are distributed throughout the crossbar switch matrix, at the interface to the memory modules leads to the multiport memory organisation, as shown in Figure 2.20, where every processor has a private bus to every passive unit, i.e. memory and I/O units. The multiple ports of every passive unit, one for each connection to a processor, are assigned fixed priorities through which arising conflicts are resolved.

This organisation offers a high potential transfer rate within the system at a comparable hardware complexity with that of the crossbar switch except for the localised logic, but with a severe constraint on the number of processors imposed by the number and type of the memory ports.

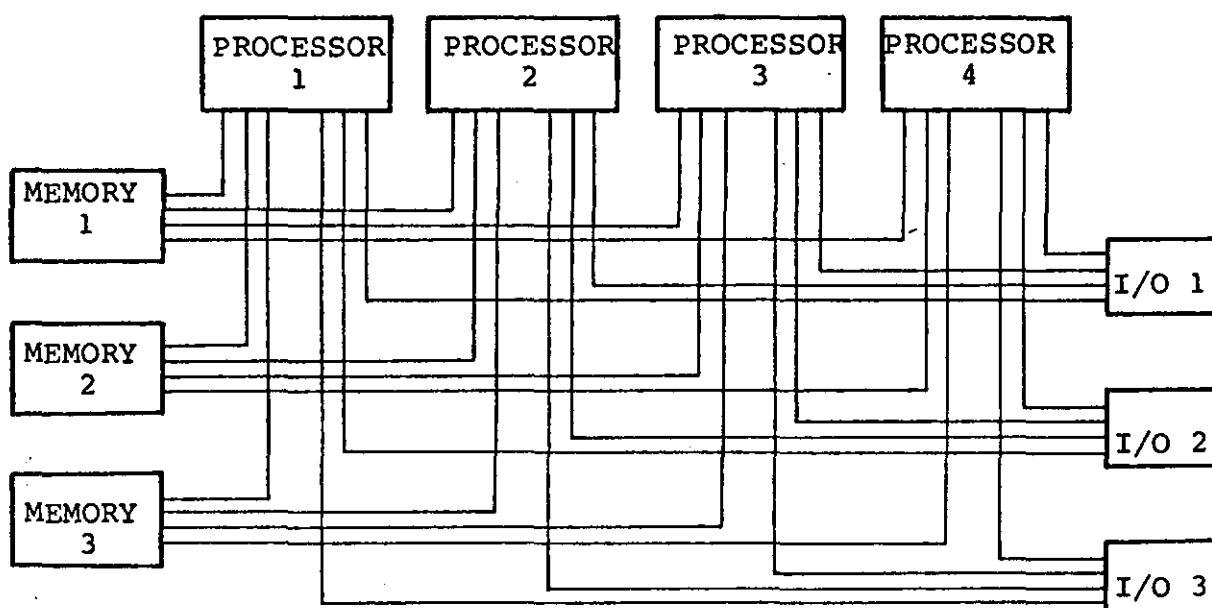


FIGURE 2.20: THE MULTI-PORT MEMORY INTERCONNECTION SYSTEM

Besides these three presented interconnection networks, there are many others which can be valuable for the multiprocessor organisation such as the Omega network [Lawrie 1975] and the Delta network [Patel 1981] and the Augmented Data Manipulator [Siegel 1979].

The interference or conflict, produced in the accessing of a shared memory, in a multiprocessor system, which is one of the factors that degrade the overall performance of the system has been investigated extensively, resulting in some exact and approximate models under various assumptions [Chang 1977], [Janek 1981], [Janek 1982], [Lillevik 1984] and [Basket 1976]. These interferences can be generally classified into two types: software and hardware types.

The first memory conflict is caused by a processor attempting to use a data set while it is currently being accessed by another processor which has eventually activated a software 'lock' mechanism to prevent any other processor from accessing the same data set. Thus, although this action forces serial manipulation of some sensitive data sets through a software mechanism, called **critical region** (to be described later on) it ensures data integrity in a multiple processor environment.

On the other hand, the second type of memory conflict is caused when two or more processors attempt to access the same memory module simultaneously, i.e. more than one request is made to the same module during a single memory cycle by different processors. Therefore, all but one request must wait to be served sequentially since only one access can be made per memory cycle. Thus, programs with a large number of these conflicts have greater degradation in their overall performance.

A way to reduce the processor interconnection network and the interference in the memory is to have a cache memory associated to each processor. The main difficulty with this approach is the coherence problem that appeared when shared data is present simultaneously in several caches. Another solution to this problem is to partition the physical memory into local memories while keeping the uniform access at the virtual level. To reduce even further the cost of the interconnection network, it is useful to divide the processors into clusters and have a slower interconnection between clusters. This approach is implemented in the Cm* [Gehrig 1982].

2.3.2 OPERATING SYSTEM ORGANISATION

Primarily, the two software support tools required for the MIMD multiprocessor systems are similar to those required for sequential computers - namely the Operating System and the general programming system. In such a system, the efficiency of these software systems is very significant, otherwise a poor performance could destroy any cost-performance advantages that the system has gained through its hardware organisation. Thus, in order to complete the whole discussion about the MIMD architectures which previously started with some hardware organisational issues, this paragraph includes a brief discussion of some basic organisations of the operating systems while Chapter 3 deals with the parallel programming issues to fully exploit the inherent parallelism in the MIMD structures.

Conceptually, there is little difference between the system software requirements of a multiprocessor and a time-shared system (i.e. using the multiprogramming concept). From the most common functional

capabilities required in the operating system, such as resource allocation and management, table and data set protection, prevention of system deadlock, abnormal termination, I/O load balancing, processor load balancing, and system reconfiguration, only the last three are considered to be unique or substantially different for multiprocessor operating systems. More specifically, the presence of more than one processing unit in the system introduces a new dimension into the design of the operating system which can be visualised in the organisation and operation of the operating system with respect to the multiple processors.

In the design of multiprocessor operating systems at least three fundamental organisations were utilised, the master-slave, the separate executive for each processor, and the symmetric or anonymous treatment of all processors.

The master-slave type of operating system which can be found in most of the earliest multiprocessor systems is, by its nature, the easiest to implement, the simplest to operate and may be derived from a uniprocessor operating system with multiprogramming facilities by including relatively simple extensions. However, this type of system is quite inefficient in utilising and controlling the system's resources. In addition, the master processor can become a bottleneck under a heavy load, consequently, many slave processors could remain idle for longer periods since the master would not be fast enough to keep them all busy and this results in a poor performance.

In this organisation the slave is restricted to perform only the user's code while the master can execute both the executive and user's code. Since only one processor is privileged the executive

code needs neither to be replicated nor re-entrant, a fact which minimises table conflicts and lock-out problems for control tables. In the case when a slave wishes to use a service that is only provided by the executive, it must first signal its intention and then wait until the master is interrupted and the executive dispatched.

With this type of operating scheme, the entire system is subject to catastrophic failures as a result of a malfunction in the master processor. On the other hand, this organisation which requires simple hardware and software, is most effective for special applications with work load well defined or for asymmetrical systems with slaves having less capabilities than the master processor.

If some of the supervisory code is made re-entrant and replicated to provide separate copies to each processor which can execute its own executive needs then a separate execute organisation is obtained. Consequently, each processor or executive has its own set of I/O equipment, files and private control tables. However, conflicts are not completely eliminated since there are some control tables which need to be shared by the entire system. Unlike in the previous organisation, the entire system remains operational in the case of a processor failure which can be restarted, although probably with some difficulties, by the operator.

The ultimate sought after multiprocessor operating system and the most complex mode of operation is perhaps more closely approached by the symmetric organisation where all the processors as well as other system resources are treated equally. In other words, all the processors with a floating master ownership are considered as an anonymous pool of resources, each of which is capable of executing a

supervisory routine as and when required. This type of system can achieve a better load balancing over all types of resources while resolving the service request conflicts through the use of priorities. Since the same service routine might be simultaneously executed by several processors, most of the supervisory routines are made re-entrant. The unavoidable table access conflicts and table lock-out delays due to the presence of multiple executives are controlled in such a manner as to preserve the system's integrity. Compared with the previous schemes, the advantages of this type of operating system are the better availability of a reduced capacity, true redundancy, the most efficient use of all the resources and graceful degradation. All but the last one are self-explanatory. A graceful degradation is the ability to reconfigure a viable system from the only remaining operational components in the case of a malfunction in some of the others.

Recapitulating, three different organisations of the operating systems for multiprocessors were presented and they all, the master slave excepted, do not constitute a "pure" example of any implemented multiprocessor operating system. In fact, most of the commercial and experimental architectures have adopted a "hybrid" approach combining all the advantageous features from all of them.

2.3.3 IMPLEMENTED MIMD MULTIPROCESSOR SYSTEMS

In this paragraph, we shall briefly present the characteristics of some of the implemented multiprocessor systems. Certainly there are several commercial and experimental multiprocessor architectures that have been developed by various manufacturers for different purposes. Some commercial ones essentially consist of extensions of a uniprocessor architecture to improve speed and reliability such as

the IBM 370/168 MP, the CDC CYBER 170 and the Burroughs B7700 (see [Satyammarayanan 1980]). Computers specifically intended for multiprocessor operations include Denelcor HEP [Smith 1981], CDC AFP [CDC 1980] and INTEL iPSC [Intel 1984] which is based on the Cosmic Cube developed at Caltech. Examples of experimental systems are the C.mmp [Wulf 1972] and the cluster of microprocessors, the Cm* [Gehrig 1982], both developed at Carnegie-Mellon University, USA.

In the Computer Studies Department at Loughborough University (UK) a group of researchers have been actively involved in an extensive multiprocessing research program leading to the development of two experimental systems, the Interdata Dual Processor and the Neptune systems. A third system, the Balance 8000 developed and commercialised by Sequent Inc, USA, was recently acquired for the development of cost-effective parallel software. In the following paragraphs, we present the different hardware and software characteristics of all these three Loughborough sited parallel systems.

2.3.3.1 THE INTERDATA DUAL PROCESSOR

The Interdata Dual Processor which was the first MIMD multiprocessor system developed in this Department can be classified as an asymmetric loosely-coupled system with very limited capabilities such as the small number of processors, the small size of shared memory, 64 Kbytes, the lack of any memory protection and the poor quality of the used software.

Initially, this system appeared as Interdata model 55 dual communications processor [Model 1971] which was subsequently upgraded by the substitution of the I/O processor (B), an Interdata

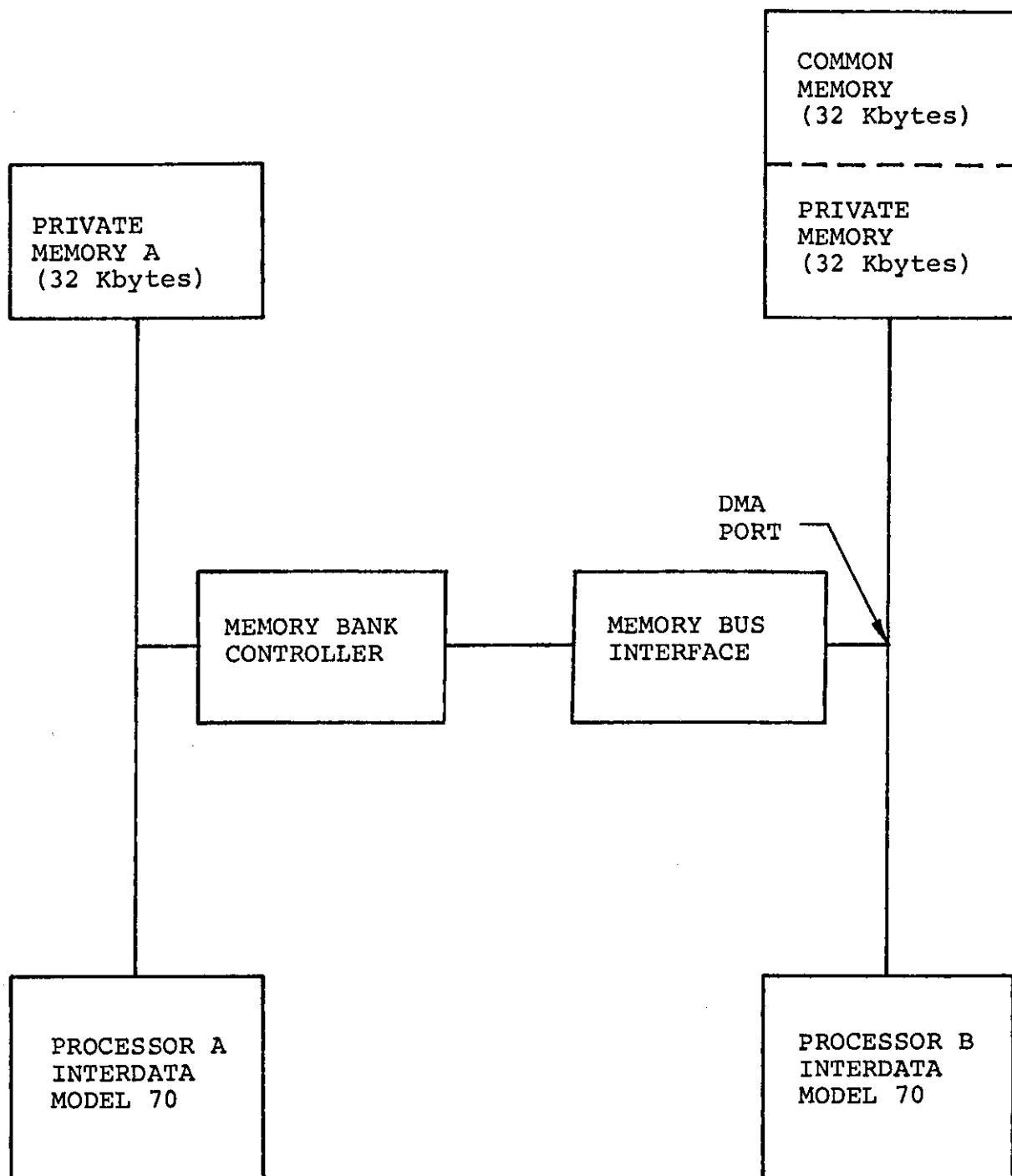


FIGURE 2.21: THE INTERDATA DUAL PROCESSOR CONFIGURATION

model 50 processor by an improved model, the Interdata model 70 processor.

Each processor of this twin system, as illustrated in Figure 2.21, is a 16-bit processor, utilising 16 registers and having private access to a 32 Kbytes local memory. The local memory of processor B was expanded to provide 32 Kbytes of a virtual shared memory, though physically processor B has a memory address space of 64 Kbytes. Consequently, the shared memory access overheads (static or dynamic) are not symmetric between the processors, a fact which justified the asymmetric property. When accessing the shared memory, processor A is delayed by 1 to 1.25 sec by the memory bus interference to B's direct memory access port, while processor B experiences no such static delay. The dynamic overheads between these processors are also asymmetric since the access of the shared memory by one processor would 'lock-out' the other one for a complete memory cycle (i.e. 1 sec), in the case of processor B, it is locked-out from both, the local and shared, memories. Furthermore, the reservation of the shared memory by processor A at least 0.5 secs before it is actually utilised (due to the memory bus interface logic) makes the dynamic overhead more asymmetric and consequently while A is dynamically delayed by .5 secs B is delayed by up to 1.5 secs.

The programs developed to run on this system can run on either processor or on both of them by initially loading common data in the shared memory and replicating the executable code in the two local memories. This twin system operates on an IBM 360 like instructions set as provided by the Interdata manufacturer. Instructions can be 16 or 32 bits long and take one or two secs to load from memory. The hardware implemented floating-point functional unit allows fast computations of arithmetic operations.

Concluding, the design simplicity of the Interdata Dual Processor along with probably small financial support has led to the development of a multiprocessor system with severe limitations, however valuable experience was gained and directly reflected in the design of the subsequent system.

2.3.3.2 THE NEPTUNE PARALLEL COMPUTER

The Neptune system, yet another system developed in this Department in 1981, (see Barlow et al [Barlow 1981]), is a homogeneous general MIMD multiprocessor system comprising four Texas Instrument 990/10 minicomputers. Since this system was used extensively in the conduction of part of the experimental work presented in this thesis, we shall examine its hardware and software characteristics in more detail. Some other features concerning this system, such as the related programming concepts and the system performance measurements are presented in Chapter 3. The physical organisation of the Neptune system is shown in Figure 2.22.

There are two types, though physically identical, of connection buses, called TILINES used in the Neptune system, four of which are utilised as local connections to the corresponding processors and directly coupled to the fifth shared TILINE in order to provide access to the shared resources (memory and disk). Each local bus connects the processor to its local memory with a capacity of 128 Kbytes optionally expandable to 512 Kbytes, except for processor 0, where in addition to the increased local memory (384 Kbytes), two 5 Mbtes of disc drives, one fixed and the other exchangeable, are also linked. A controller with 474 Mbytes Winchester disk drive as well as a magnetic tape streamer are connected to processor 2. The

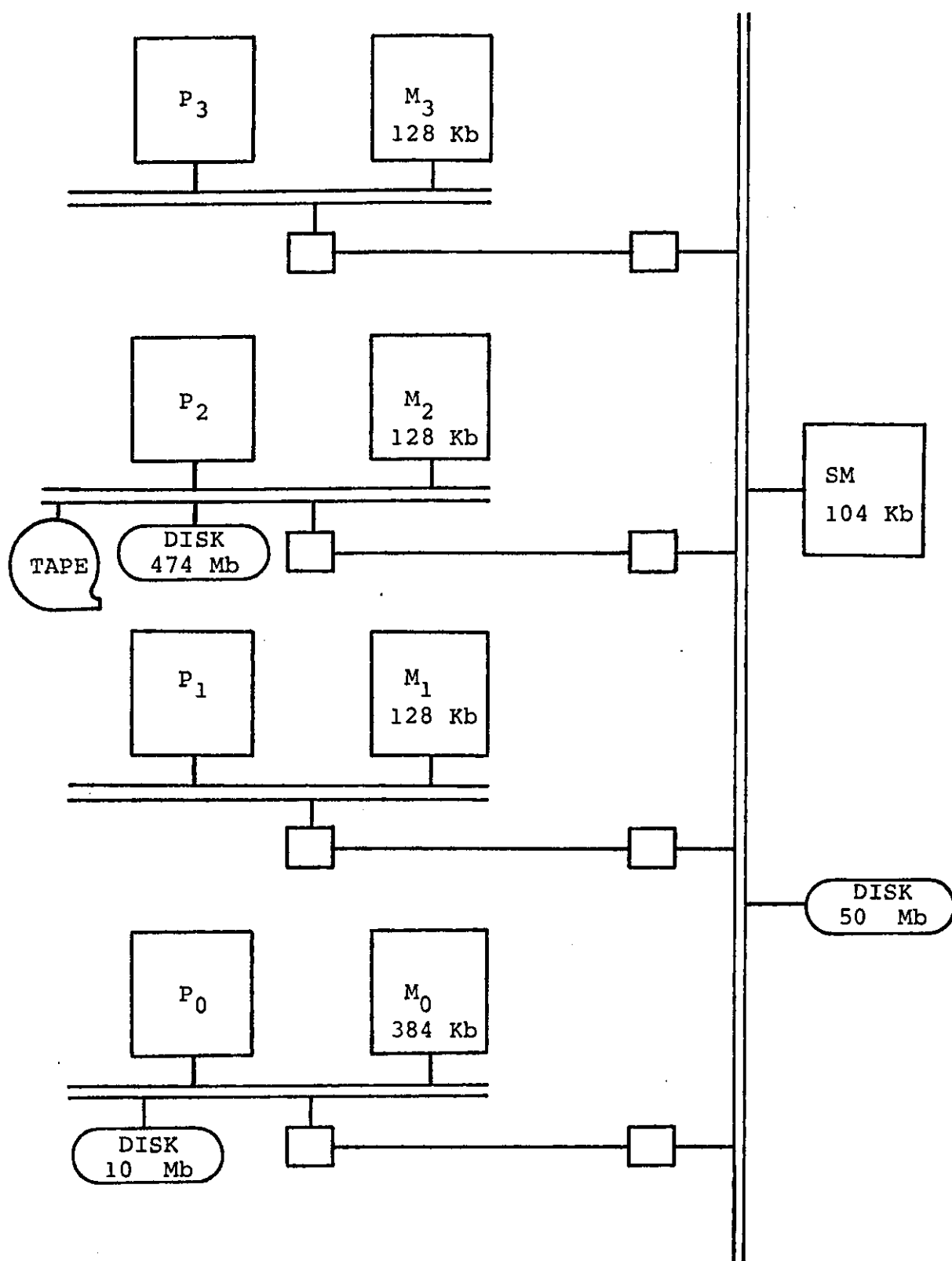


FIGURE 2.22: THE CURRENT NEPTURE SYSTEM CONFIGURATION

shared TILINE connects all the four processors' local buses via a TILINE coupler to the 104 Kbytes of shared memory and 50 Mbytes disc.

Generally speaking, in a fully symmetric and homogeneous MIMD multiprocessor system one should allow some fluctuations in the performance of the available processors and does not always expect an exact performance figure for all processors mainly because of the changing atmospheric conditions which affect slightly the efficiency of the cooling system and also because of the unpredictable dynamic behaviour of the system. This is unofficially verified by the processors' performance measurements as carried out by the staff supporting the system. More specifically, the time for each processor to access its local memory is approximately .6 secs, whereas the shared memory access time is 1.41, 1.12, 1.31 and 1.32 for processors P0, P1, P2 and P3 respectively. Consequently, the following relative processor speeds of 1.000, 1.037, 1.006 and 0.978 were measured for P0, P1, P2 and P3 respectively, a fact that reduces the efficiency and decreases the performance of parallel algorithms, especially those with synchronisation [Barlow 1981].

Logically, the organisation of the Neptune system is such that the multiprogramming (the simultaneous co-existence of several programs) and the multi-tasking (the cooperation of several processors for the completion of a single task) modes of operation are possible but under the supervision of the user. Thus, during normal use, this system operates like four individual processing systems, each of which support multiprogramming. In this case, the shared memory or part of it can be requested by the processors in order to increase their own local memory storage. In the second mode of operation, once processors are allocated, to usually the first bidder, they are

locked-out preventing any other parallel task to execute until the completion of the current task. Because of the full symmetry observed in such a system no single processor can potentially limit the overall performance and in fact the pool of the processors works as a team under all the conditions to maximise the system's efficiency.

Commands used in the development of parallel programs specifically ensure that these programmes are split into two parts, one containing the program code and the local data and the other one containing the shared variables, which must internally reside in two separate segments. Now, a processor receiving a request to execute a parallel task, as such, this processor would be known as the **initiator**, must first claim sufficient space in the shared memory to load the segment containing the shared variables once its request has been granted. Subsequently, the management area is set to contain pointers to that shared segment and tasks are activated in the remaining requested processors with enough information to load the segment containing the local variable into their respective private memories before starting the execution of the intended parallel program. Except for the initiator, all the processors have a **link** to the shared segment which resides in the common memory.

All the Neptune's processors operate under the control of the powerful **DX10** uniprocessor operating system which is a general-purpose system with several enhanced and sophisticated features to support multi-tasking. This system features several effective packages which are subsequently presented, though in a very brief manner and the interested reader is referred to the Texas Instrument manuals [Texas] for more detailed information about any facility of the Operating System **DX10**.

The DX10 Operating system provides an effective tree-structured filing management system which supports multi-indexed files. The specification of a file takes the form of a succession of directory names encountered when travelling down the tree, starting from the root which can be a specific disk pack or volume until reaching the actual leaf or filename. In the case where the list is too long and becomes cumbersome, the synonym facility can be utilised to replace this long string of characters by a shorter one. For example, a directory USER.PG.DIR1.DIR2 can be replaced by the synonym DIR, which means that all files belonging to this directory can be referred to as DIR.filename instead of USER.PG.DIR1.DIR2.filename. The system will automatically evaluate the given synonym before any file operation. Additionally, the volume name can be omitted in the case when the file resides in the system volume. Such a system must be carefully designed to include some coordination allowing shared files to be created, opened, read, written, closed as well as deleted by more than one concurrent process. However, the standard DX10 limitation for only one task with the file open for writing as well as the lack of direct updating (i.e. the users of the shared file are notified of any change in the file only when the writer closes the file) are still restricting the simultaneous accesses to a single file.

From the user's point of view, the 'System Command Interpreter' (SCI) provides a friendly user interface to the system by means of displayed menus and a comprehensive prompting system, which assists in entering commands and their eventual parameters. The SCI which can be involved interactively or through a runnable program includes a check of all the given values. Besides the SCI, two additional features, the foreground and the background facilities are available

during the execution of a task. The former, which can be owned by only one user to execute only one task at any given time, automatically suspends the SCI as soon as it is invoked. On the other hand, the latter which is a multi-tasking management environment, can be invoked while the SCI is still available to process the user requests. Consequently, the state of these background tasks can be inspected at any time through the interrogating SCI commands.

The frequent hardware and software alterations or extensions that the Neptune system has to experience is bound to increase the rate of malfunctioning problems. The most common known cause for a malfunction, known as a system crash is when one or more processors fail, indicated by a fault on the front panel. A manual procedure is provided on the front panel of each processor and can be used to reload the system after dumping its contents for later crash analysis.

In order to recover from an eventual system catastrophe that could destroy all or part of the important files, the Neptune system provides a dumping mechanism, in a short (daily except Sunday) or long (monthly) term basis.

Finally, several modifications have, however, been underway for the DX10 to produce a new version (DX10 Mk 3.5) along with the instalment of new hardware equipment (such as memory, hardware floating point, resource management) and the development of some new facilities such as a new preprocessor to use on the VAX, a file transfer protocol between VAX and Neptune and the implementation of a PASCAL-PLUS compiler.

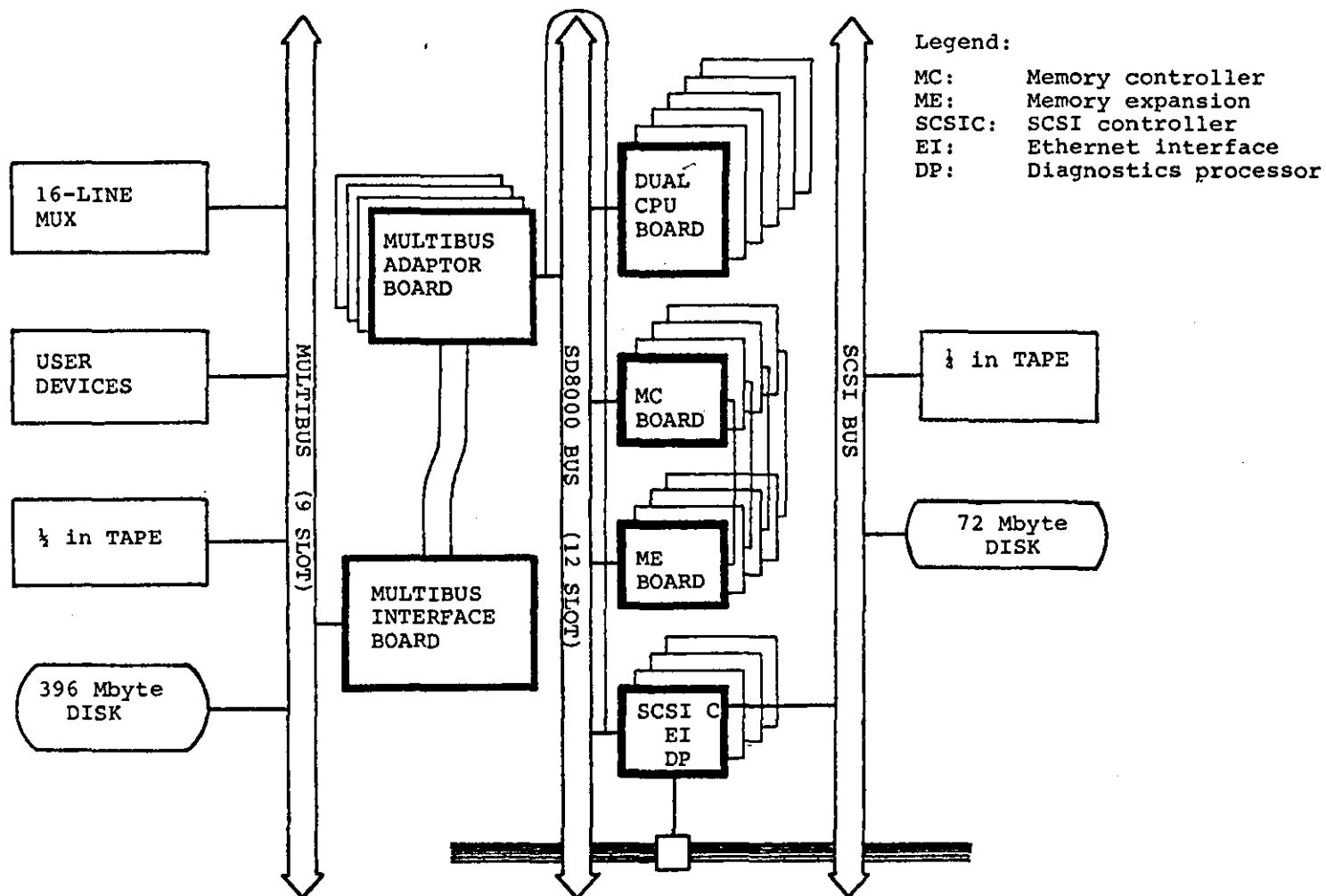
2.3.3.3 THE SEQUENT BALANCE 8000 SYSTEM

Recently, a third system, the Balance 8000 which was developed by Sequent Computer Systems Inc, using a new processor pool architecture was installed in the Computer Studies Department. This system dynamically shares its load among twelve architecturally similar processing units and operates under a single copy of a Unix-based operating system, known as DYNIX, capable of delivering up to 5 MIPS. The pool processing organisation requires dynamic balancing of the system workload among the processors with an effective use of all resources in general. Consequently the system automatically and continuously assigns tasks to run on any processor that is currently idle or busy with a lower priority task, meaning that a process does not necessarily run to completion on the same processor but on the contrary it may involve several processors. This balancing process is carried out transparently; neither the user nor the programmer need to be aware that the system supports multi-tasking operations.

From the hardware point of view, the Balance 8000 consists of a pool of two to twelve processors, a bandwidth bus, up to 28 Mbytes of main memory, a diagnostic processor, up to four high-performance I/O channels and up to four IEEE-796 (Multibus) bus couplers. Figure 2.23 shows the main functional blocks of the Balance 8000 System.

Each processor is a subsystem containing three VLSI components: a 32-bit processing unit, a hardware floating-point unit and a paged virtual memory management unit. Two such subsystems are on one circuit board (see Figure 2.24 which shows the major units of a dual processor board). Also each processor contains a cache memory that almost reduces to zero all the processor waiting periods and minimises the bus traffic. The two-way set-associative cache

FIGURE 2.23: THE BALANCE 8000 SYSTEM CONFIGURATION



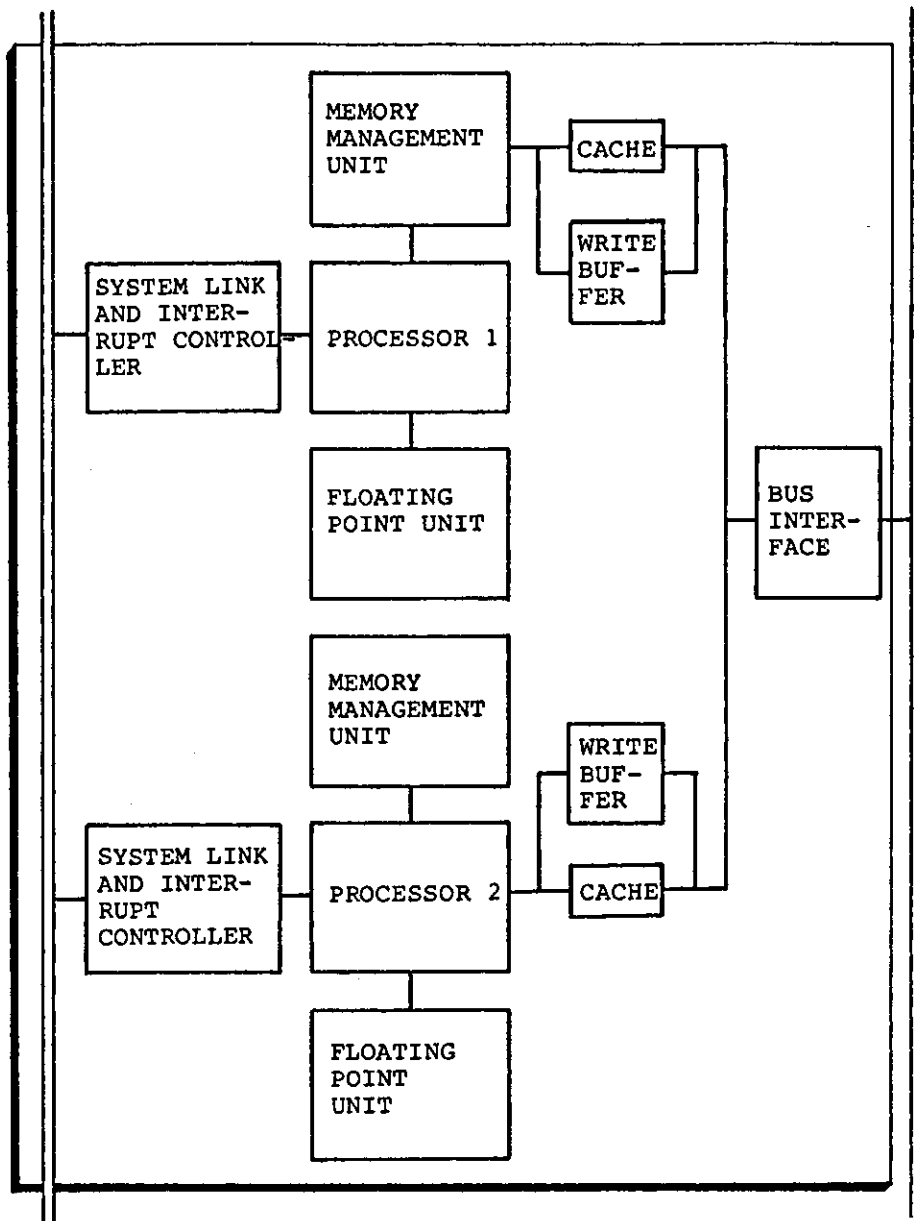


FIGURE 2.24: CONFIGURATION OF A CPU BOARD WITH 3 VLSI COMPONENTS ATTACHED TO EACH PROCESSOR

consists of 8 Kbytes of very high speed memory and stores recently accessed instructions and data, so subsequent requests for the same data are satisfied from the cache, rather than from the main memory.

However, with the use of these cache memories two coherence problems arise, mainly the coherence of the data between the main memory and the caches on each processor and the coherence of the data between the caches themselves. For the former problem, a **write-through** mechanism is utilised in order to keep the main memory up to date with all the eventual changes made in every processor's cache. In addition to the update of the appropriate cache, this mechanism would allow the same write cycle to pass to the bus and memory. In the latter case, the answer is provided by the bus watching logic implemented in every cache. Consequently, all the write cycles on the bus are monitored and the addresses are compared with those in the cache, so whenever the contents of the cache are altered, the cache invalidates the entry in question.

Significant processing time is saved by including a **write-buffer** in each processor which can proceed immediately after issuing a write cycle letting the buffer wait for the memory cycle to complete.

Finally, to complete the description of the components found in the processor subsystem we need to refer to the 'System Link Interrupt Controller' (SLIC) which is a chip, one for each processor and for every other board, attached to the SB8000 bus. This SLIC chip manages interprocessor communication, synchronised access to shared data structures, distribution of interrupts among the processors, and diagnostics and configuration control. The SLIC bus which is a part of the SB8000 system bus provides an interconnection for communication among the SLIC chips.

The SB8000 system bus is a 32-bit wide, pipelined, packet bus supporting multiple overlapped memory and I/O transactions and capable of achieving a throughput rate of 26 Mbyte/sec. It also supports several packet lengths and checks parity to aid in error detection.

This system provides up to 28 Mbytes of principal memory, a 4 Mbytes I/O address space that can be shared by all the processors and a 16 Mbyte virtual memory address space for each process. The Balance 8000 supports up to four memory controllers, each with an optional expansion board, reducing memory contention among processors. It also supports standard I/O throughout the system, and permits several instances of each interface to increase the I/O bandwidth. More specifically this system supports a SCSI interface for disc and tape I/O, a Multibus interface for serial communications, large disc and tape support, and user-added devices, and finally an Ethernet local area network for communication among systems.

From the system's software point of view, this system operates under the powerful DYNIX which is based on the UNIX uniprocessor operating system with several significant enhanced features to support multi-tasking. The DYNIX Kernel or executive has been made shareable so that all the processors can execute the same system calls and other kernel code simultaneously. The DYNIX system schedules the processes to execute on the processors such that the workload is well balanced. This means that any user or system defined process can run on any processor at any time and may involve several processors to complete. The DYNIX determines the minimum and maximum amount of physical memory that a given process can consume, then adjusts the memory allocation for each process between these

two bounds to maintain each process's paging rate and tune the virtual memory performance for the entire system.

Full advantage is taken of the UNIX filing system and the multiprogramming features such as **pipes** and **forks** that are automatically executed in parallel. The Dynix and the parallel programming library supply the fundamental parallel programming mechanisms such as **process** creation and **termination**, interprocess communication and synchronisation via the shared memory and UNIX signals and mutual exclusion via **spinlocks**.

Chapter 3

PROGRAMMING TOOLS AND PERFORMANCE ANALYSIS OF PARALLEL ALGORITHMS

3.1 PARALLELISM DETECTION

It is certainly true that while the computer architecture - in particular the advances generated by LSI - is bringing this new revolution in computing, the programming tools to fully exploit the potential parallelism are only slowly forthcoming. Realising the serious consequences that are likely to result from any mismatch between the hardware and the software, the computer researchers have oriented much of their efforts towards parallel software engineering development.

Obviously, any parallel system is considerably more difficult from the programming point of view than a conventional uniprocessor computer since the designer is faced with additional complex decisions to make so as to balance the problem requirements against the available resources. Although the process of making these decisions in order to develop effective parallel software for a particular parallel system is still an ad-hoc procedure, the accumulation of all these individually gained experiences could well shed some light on how effective various strategies are at exploiting parallelism.

There are at least three emerging parallel software design approaches based upon the concealment (or not) of the parallelism by the hardware structure. In other terms, for some architectures, the parallelism is hidden by the hardware itself whilst for others it is revealed to the user so that appropriate decisions are made as and when needed.

The first of these approaches, the automatic translation of sequential programs or the implicit parallelism, which is outlined in Section 3.1.1, relies on sophisticated compiling techniques to partition a global task written in a high-level sequential language. With the use of this approach, it is hoped to take advantage of the huge amounts of existing sequential software. For example, a sequential program could be used to generate several versions of parallel algorithms, each suitable for a particular type of parallel computer (pipelined, array, multiprocessor, etc). Consequently, the complexity of writing algorithms is no worse than that of a uniprocessor system. In addition, if the compiler has been fully debugged, then the program decomposition is correct by construction. The disadvantage of such a method is the complexity of the compilation task that makes the approach unsuitable for most programs that are run only a few times. Also, since this approach was proven for rather simple numerical applications, for more sophisticated applications, such as non-numerical algorithms, there are doubts that it will be successful.

The second approach, which is considered in Section 3.1.2 is explicit parallelism. The programmer manages the concurrency of the application by coding directly in a concurrent language (e.g. ADA or CSP - 'Concurrent Sequential Processing') or in a high-level language with many embedded parallel constructs. Both types of languages have special statements for tasks initiation, termination, synchronisation and message passing that allow efficient coding of even more sophisticated applications. One of the most significant advantages of such an approach is that the actual architecture characteristics are taken into account so as to generate efficient parallel algorithms. Consequently a better match between the hardware and software could be obtained in order to achieve the

intended design goals that the system was first built for. For example, the algorithms, designed for an Array Processor, must be developed to keep every processing element as busy as possible to achieve a high-degree of parallelism. Therefore, in such a system, we are not primarily interested with the efficient use of the array processors but rather by the speed-up factor. On the other hand, in a MIMD multiprocessor computer, and due to the asynchronous nature of the processors, if a processor has little effect on the run-time of the algorithm, it is better from the processors efficiency point of view to use it elsewhere on a different task (i.e. task rebalancing). Thus, in an MIMD computer, the concern is with the efficient use of the processors coupled with the speed at which the problem is solved. Due to the concurrency problem, these programs are significantly more difficult from the debugging point of view. Thus, it is a complex task to track down an error in a concurrent program.

The third approach, advocated by Backus and Dennis, is based on the functional language model (see [Backus 1978] and [Dennis 1966]) and is implemented on most data flow computers. Relying on the programmer's ability, the former method could rapidly become unworkable as it is impossible to keep "juggling" with a large number of tasks. The functional approach, which is the most natural form of handling parallelism can achieve the highest degree of concurrency since the instructions are scheduled for execution directly by the availability of their operands. However, the high cost of implementing this unstructured low-level concurrency makes this method of less importance, at least for the present moment.

In the remaining sections of this chapter, we shall be concerned with the structure of parallel algorithms, presenting the necessary

parallel constructs used when implementing such programs, and finally by the performance analysis of this class of parallel algorithms as adopted in the Computer Studies Department at Loughborough University.

3.1.1 IMPLICIT PARALLELISM

Most of the existing sequential software exhibits naturally some form of concurrency which needs only to be identified and then exploited in the design of parallel algorithms. One of the approaches to parallelism that relies on the implicit detection of parallel processable tasks within a sequential algorithm is the implicit approach. Several sophisticated compiling techniques were developed to automatically translate a sequential program into a form suitable for parallel processing on a particular type of parallel machine. In addition, such a process must also determine the dependency relationship among the various identified tasks so as to effectively schedule them for parallel execution.

Several automatic recognition schemes, some of which are subsequently presented, have been proposed to accomplish this detection. We should emphasise at this point that none of the presented schemes can be universally implemented since it is dependent on the source language.

In 1966, Bernstein developed the deterministic conditions which were sufficient for the parallel execution of sequentially organised processes (see [Bernstein 1966]). His proposed detection method, presented in terms of sets representing memory locations, is based on four different ways of utilizing a memory location by a sequence of instructions or tasks. These four conditions are:

1. The location is only fetched during the execution of a task
2. The location is only stored during the execution of a task
3. The first operation within a task involves a fetch, with respect to a location. One of the succeeding operations stores in this location
4. The first operation within a task involves a store with respect to a location. One of the succeeding operations fetches this location.

Although these conditions were sufficient to ensure the commutativity of two tasks that can execute in parallel, they are very poor in deciding these factors when presented with arbitrary programs. This work was complemented by that of Fisher who presented an algorithmic implementation of the above conditions [Fisher 1967]. In this algorithm, the input and output sets of each task were used to determine the required ordering and thus the inherent parallelism.

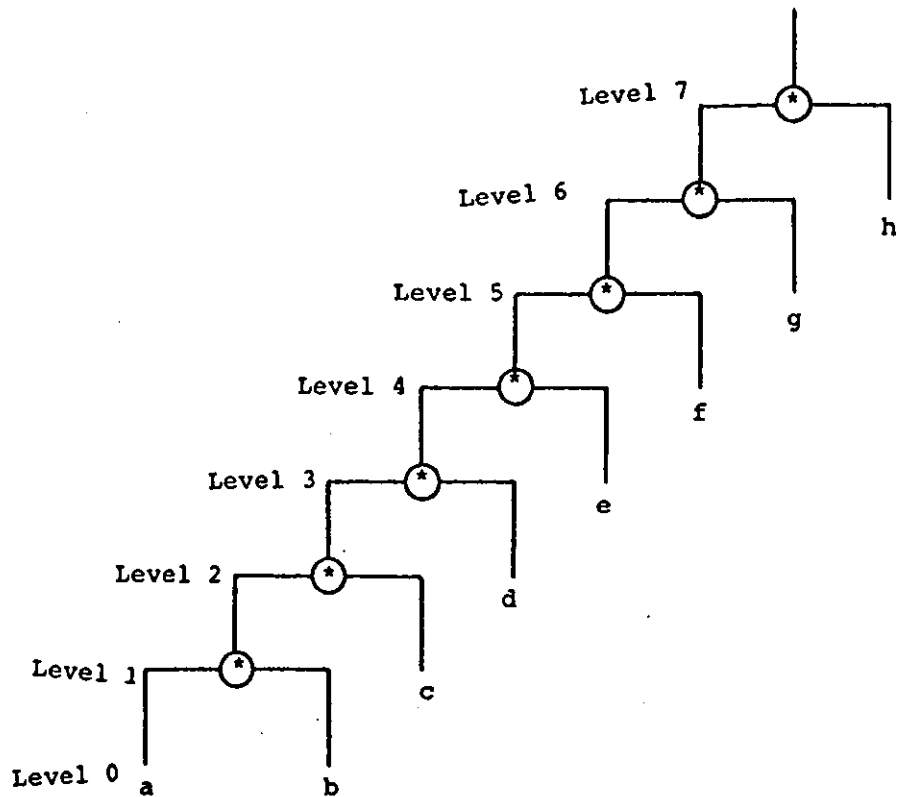
In 1969, Ramamoorthy and Gonzalez presented a Fortran Parallel Task Recognizer using a new approach based on the computational modelling of the processes using oriented graphs (see [Ramamoorthy 1969] and [Gonzalez 1969]). In these graphs, the nodes (vertices) represented single tasks and the oriented arcs (edges) represented the allowed control sequencing of the tasks. Thus, the processes properties could be investigated by simply manipulating the corresponding connectivity matrix of the considered graph.

In 1978, Evans and Williams [Evans 1978] introduced a method of detecting parallelism in ALGOL-type programs, providing the required translation code for many particular language constructs such as loops, conditional branches and assignment statements. The

implementation of these constructs was performed by Williams in 1978, who presented an ALGOL 68-R program describing how a multi-pass compiler can detect the potential parallelism. This compiler was subsequently extended to include two more stages, the Analyser and the Detector programs. The role of the analyser was to partition a given program into logically independent tasks, such as sub-programs, loops etc, which are then examined by the detector to determine whether there is a parallel relationship between them.

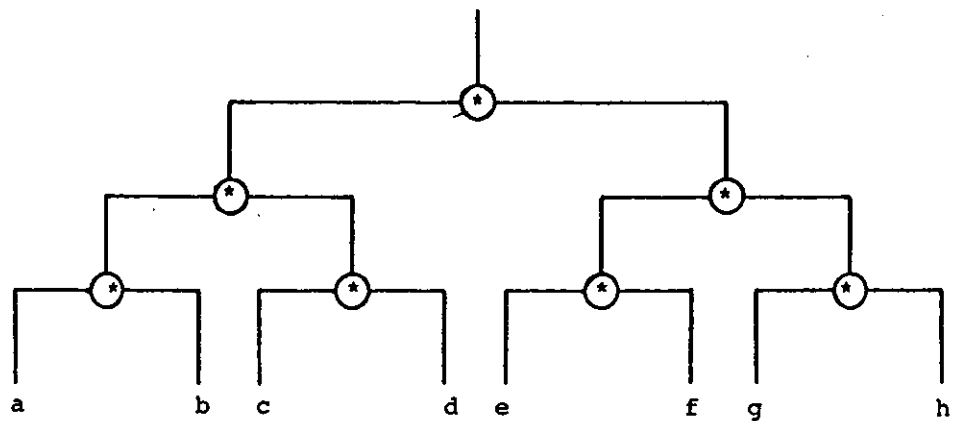
One of the most studied detection schemes that has been given much consideration is the implicit detection of the inherent parallelism within the computation of arithmetic expressions. Because of the sequential nature of most of the uniprocessor systems, the run-time of any arithmetic expression computation is always proportional to the number of operations. This run-time can be further reduced on a parallel system by concurrently processing many parts of the expression. In fact, the commutativity and the associativity were extensively used in order to reduce the height of the computational tree representation. For example, the expression $(a*b*c*d*e*f*g*h)$ can be rearranged in a form suitable for parallel processing $((((a*b)*(c*d))*((e*f)*(g*h))))$. As it can be seen in Figure 3.1 which depicts the tree representation of the above expression for a sequential and a parallel processor respectively, the run-time was reduced by four time units.

There is much literature about algorithms dealing with the detection of parallelism at the arithmetic expression level, some of which are those proposed by Squire [Squire 1963], Hellerman [Hellerman 1966], Stone [Stone 1967], Baer and Bover [Baer 1968], Ramamoorthy and Gonzalez [Ramamoorthy 1969], and Muller and Preparata [Muller 1976].



(i) Serial computer

FIGURE 3.1: TWO POSSIBLE BINARY TREE REPRESENTATIONS OF THE EXPRESSION $a*b*c*d*e*f*g*h$ FOR A SERIAL AND PARALLEL COMPUTER RESPECTIVELY



(ii) Parallel Computer

By all means, this is not intended to be a complete survey of all the proposed methods, but only an attempt to emphasize the major interest in this area. However we should complete this presentation with the work of Kuck [Kuck 1977] and Wang and Liu [Wang 1980].

In some particular cases, the use of the commutativity and associativity properties does not always lower the height of the computational tree. Kuck studied the effectiveness of the application of the distribution at reducing the tree height to its minimum value. Although this technique may involve some overheads, it, however, generates a faster parallel algorithm (see Figure 3.2).

Finally, Wang and Liu introduced the concept of the 'parallel Execution String' (PES) which, unlike the previous methods, can also detect parallelism at the statement and block levels in order to maximise the amount of concurrency. They also designed two algorithms that translate arithmetic expressions into PES's.

3.1.2 EXPLICIT PARALLELISM

In this approach to parallelism, the programmer has to specify explicitly those tasks that can be performed concurrently by means of special parallel constructs added to a high-level programming language. Although these programming constructs can be time consuming and difficult to implement they can offer significant algorithm design flexibility; in other words, many different possible structures of the same algorithm can be analysed until a satisfactory version is obtained.

Considerable research has been done on this approach with a particular interest on those parallel task issues such as task

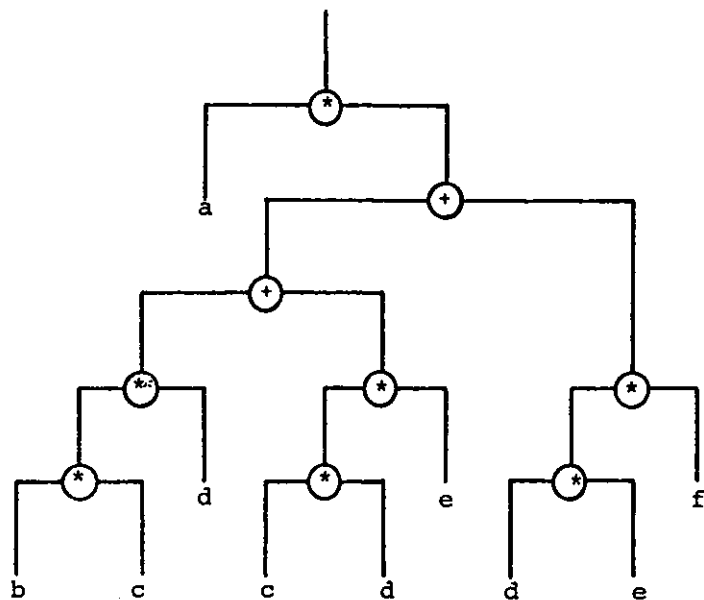


FIGURE 3.2(a): TREE REPRESENTATION OF THE EXPRESSIONS
 $a*(b*c*d + c*d*e + d*e*f)$

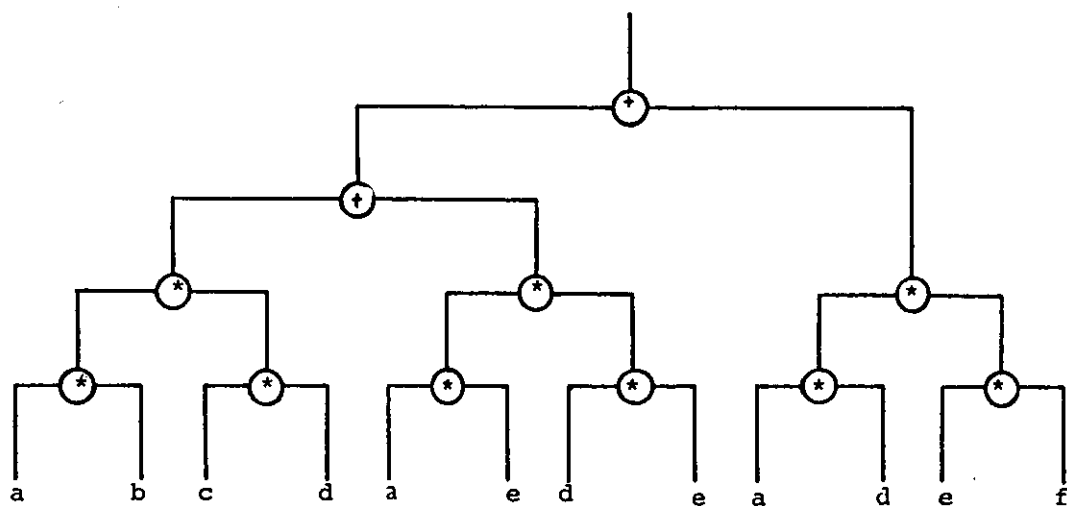


FIGURE 3.2(b): TREE REPRESENTATION OBTAINED BY 'DISTRIBUTING' a .
 THE HEIGHT IS THUS REDUCED FROM 5 TO 4

declaration, activation, termination, synchronisation and communication of which the latter two are the most significant. First, we shall present some of the synchronisation and communication mechanisms that have been proposed and then introduce the different techniques used to express concurrency.

If several concurrent processes are sharing a critical data item, then they must synchronise their operation so that at most only one of them is in control of that data (mutual exclusion). Although this process synchronisation can effectively ensure data integrity it unavoidably forces sequential handling of the shared data by the created processes.

In a paper published in 1965, Dijkstra suggested the utilisation of semaphores and introduced two new primitives (P and V) that greatly simplified process synchronisation and communication [Dijkstra 1965]. A software implementation of these two primitives in terms of an indivisible instruction, the test-and-set instruction, was installed in many systems. Although the utilisation of semaphores successfully allowed a harmonious cooperation between several processes, it has not reduced the total interference when a large number of variables are shared.

Other synchronisation primitives that serve the same functions as P and V have been suggested. For example Dennis and Van Horn [Dennis 1966] suggested a very straightforward mutual exclusion lock out mechanism. Critical regions, i.e. the set of instructions that manipulates the critical data, are enclosed within a LOCK W - UNLOCK W pair, where W is an arbitrary one-bit variable.

In general, many difficulties may arise when using these low-level synchronisation mechanisms since they do not facilitate the compiler's role in checking possible error conditions. More specifically since a semaphore can be used to solve arbitrary synchronising problems, a compiler cannot conclude that a pair of P and V operations on a given semaphore delimits a critical region, or that a missing member of such a pair is an error. The compiler will also be unaware of the correspondence between the semaphore and the common variable it protects. Thus, a compiler cannot give the programmer any assistance in establishing that a program is error free with absolute certainty.

At least three high-level synchronisation and communication mechanisms have been proposed. For instance, the conditional critical regions (see [Hoare 1972] and Hansen [1973]) and the monitors (see Hansen 1977) concepts have significantly reduced the potential amount of interference by grouping the shared variables into resources, with exclusive access to them. In the former concept, a process is allowed to test the state of a resource before entering a critical region to determine whether the corresponding operation is permissible or not, and if it is not, to wait until other processes have brought the resource into a state by which the operation is permissible. On the other hand, with the latter concept which is a language construct, the compiler is informed about the shared data structures as well as the operations (or procedures) that processes can perform on them. Thus, functionally a monitor is a collection of data and procedures operating on this data, shared by several processes on a mutual exclusion basis. The operations WAIT and SIGNAL, initially suggested by Campbell and Haberman [Campbell 1974] for synchronisation purposes, are very useful for requesting and releasing resources.

Unlike the previous two concepts which are essentially centralised facilities, the third one, the ADA-Rendezvous concept is more oriented towards message passing between processors in a distributed system environment (see [Hoare 1978]). When two processes decide to rendezvous, the first one to arrive at the rendezvous point is blocked until the arrival of the second. Many other powerful facilities are also provided in the ADA which is considered one of the most powerful languages. However, since the use of these complex constructs could lead to potentially more sharing, a special care should be taken when implementing an application program in this language.

The utilisation of the **critical section** is without any doubt the most effective way to reduce interference amongst active processes. In this concept, which is implemented in the Loughborough multiprocessor systems, the section of the program code that accesses the **critical data** is called critical section and it is executed by only one process at any time. Two operations, **\$ENTRY** and **\$EXIT**, ensure that this section is shared between processes on a mutual exclusion basis. In the case of arising access conflicts, a protocol is provided so as to schedule one of the contending processes to enter the critical section. This will be complemented by a detailed study in the following paragraph when we examine the actual implementation of some of the parallel constructs in the MIMD multiprocessors currently operational at Loughborough University.

Several mechanisms for expressing concurrency have been developed in the form of additional parallel constructs. For instance, **COBEGIN** [Dijkstra 1968] or **PROCESS** declaration [Hansen 1975] were utilised to specify those parts of the program that can execute concurrently,

distinguishing between the local variables and the shared ones. Another example is the **PARALLEL FOR** [Gosden 1966] which generates for every iteration of the ALGOL for statement a separate parallel process.

In 1965, Anderson [Anderson 1965] introduced five parallel constructs, the **FORK**, **JOIN**, **TERMINATE**, **OBTAIN** and **RELEASE** statements which are presented below in an ALGOL-68 format:

```

<Fork statement>::= FORK <Label list>;
<Joint statement>::= Label: JOIN <Label list>;
<Terminate statement>::= Label: TERMINATE <Label list>;
<Obtain statement>::= OBTAIN <Variable list>;
<Release statement>::= RELEASE <Variable list>;

```

where

```

<Label list>::= Label/Label, <Label list>;
<Variable list>::= Variable/variable, <Variable list>.

```

The **FORK** statement is used to generate as many parallel tasks as there are labels in the list, initiating the control of each one at the address specified by the corresponding label. All the labels must be locally defined, i.e. only those labels used within the block scope in which this statement is utilised. As an arbitrary parallel program (see Figure 3.3 which depicts a typical program using these parallel constructs) can include many forks at different levels, the next sequence of tasks may only be initiated when all the forked tasks of the previous level have completed their execution. However, few exceptions, such as a branch operation to alert an I/O unit for a momentary utilisation, do not have to be completed before more tasks are initiated. In some cases, it is sometimes desirable to release some of the processors without the

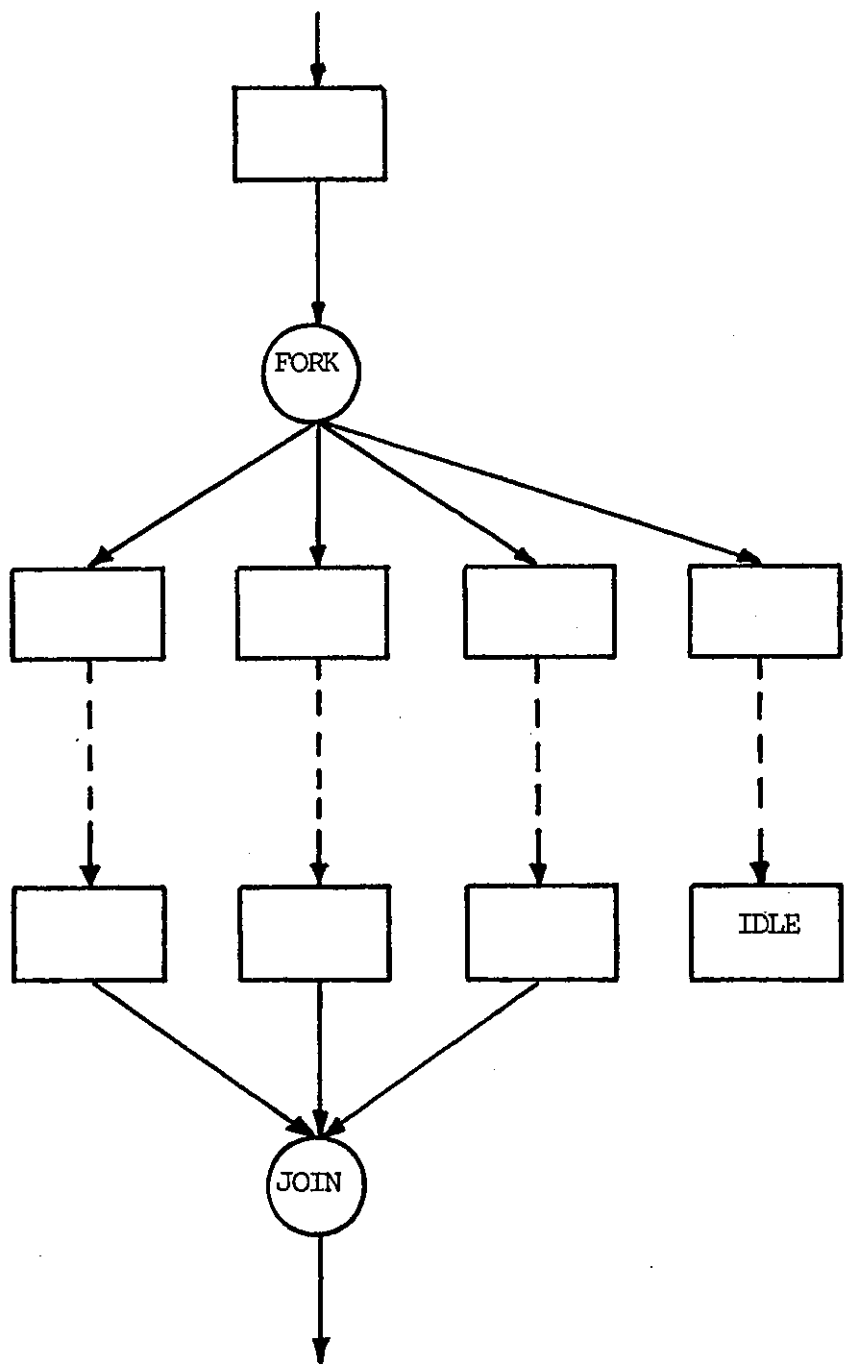


FIGURE 3.3: THE FORK/JOIN TECHNIQUE

initiation of further tasks. To achieve this, an **IDLE** statement has been proposed [Gosden 1966].

The **JOIN** statement which is closely associated with the above statement is used to terminate the parallel processes that have been forked and a single task may subsequently follow. This is implemented by including a code that causes test bits to be available, thus allowing the forked paths to be synchronised after they are completed. Every generated task must include at the end of its code a branch operation to the **JOIN** statement.

In the **ALGOL-60**, the recursive subroutine call mechanism relies upon finding a condition that allows a normal exit from this subroutine at execution time. Since the **FORK-JOIN** concept is functionally identical to a recursive call¹, it was necessary to include the **TERMINATE** instruction to explicitly de-activate some of the unnecessary tasks.

The **JOIN** and **TERMINATE** parallel constructs are currently implemented as control counters being initialised at the compilation time by the number of labels appearing in the list. When one of these two instructions is executed, the content of the counter is decreased by one and then compared to zero. If this content is greater than zero then a task has just completed and the processor is free to go and execute another pending task; otherwise, this processor has to synchronise until all the remaining (if any) ones join it.

The **OBTAIN** statement is used to provide exclusive access to the listed variables by a single process. Consequently, this mechanism

1. In fact, tasks are forked sequentially until the exhaustion of the label list.

can avoid mutual interference by locking-out other parallel tasks from the use of these variables. The multiple handling of a list of variables by several concurrent processes requires that these resources are shared or are common to all the processes, a fact which explains that all the variables used with the OBTAIN statement must be defined in higher level blocks.

The logical counterpart of the OBTAIN statement is the RELEASE statement which is used to selectively de-allocate all those no-longer required shared variables. Thus, any process waiting to access shared variables can eventually proceed if its list of variables is made available.

A similar concept to the OBTAIN/RELEASE pair is the LOCK-UNLOCK concept which was introduced by Dennis and van Horn [Dennis 1966] in 1966. One of the major difficulties of this concept when operational is the deadlock problem which can result from several different situations. The most classical are these two: (1) the pre-emption of a process with the lock activated by a higher priority computational process, and (2) when two processes try to acquire exclusive access to two shared variables but in reverse order to each other. Each is preventing the other to proceed since it is holding a variable requested by the other. Consequently neither of them can execute (deadlock). One obvious solution to the first case is to inhibit interrupts between the execution of the LOCK and UNLOCK pair whereas there seems no apparent solution to the second problem apart from detecting the deadlock and try once more.

In conclusion, all the above presented constructs are directly implemented as library functions and supplied with enough information so as to be able to generate and control parallel

activities. In particular, the FORK statement would be substituted at the compilation time by a special code that when executed would create as many parallel tasks as the number of labels following the FORK statement. Each of these tasks is assigned to the available processors and usually the first task is assigned to the processor that carries out the FORK statement itself. In the case that the number of created processes is greater than the number of available processors, the excess tasks are kept in a queue until a processor becomes free.

All the labels used in a parallel program are cross-referenced at the compilation time by arranging them on a forward reference list which is loaded by all the labels contained in the labels list of an instruction. When a label is encountered, this list is searched and if this current label is found, it is removed from the list and a special heading information (may include code length, data etc) is generated just before the labelled block.

3.2 PARALLEL PROGRAMMING SUPPORT OF THE LOUGHBOROUGH MIMD SYSTEMS

Finally, to complete our discussion about the Loughborough multiprocessing systems which was previously started by outlining their hardware and software characteristics (see Section 2.3.3), we shall present in the following sections the parallel programming concepts as implemented in such systems. The provided facilities would allow the parallel program designer to define in a simple manner the creation and termination of parallel processes (or paths), which data is shared between paths, and a reliable update operation of certain shared data structures [Barlow 1981].

For instance, let us consider an arbitrary algorithm consisting of five smaller segments, S1, P1, P2, P3 and S2 as illustrated in Figure 3.3(a). Of these segments, P1, P2 and P3 are assumed to be independent and so they can be executed in parallel. The parallelisation of this algorithm leads to the one shown in Figure 3.3(b), where after the execution of segment S1, three parallel paths are created and executed; after the completion of these paths, the segment S2 is executed. Thus, this simple example illustrates many of the important issues to be considered when a parallel algorithm is being designed.

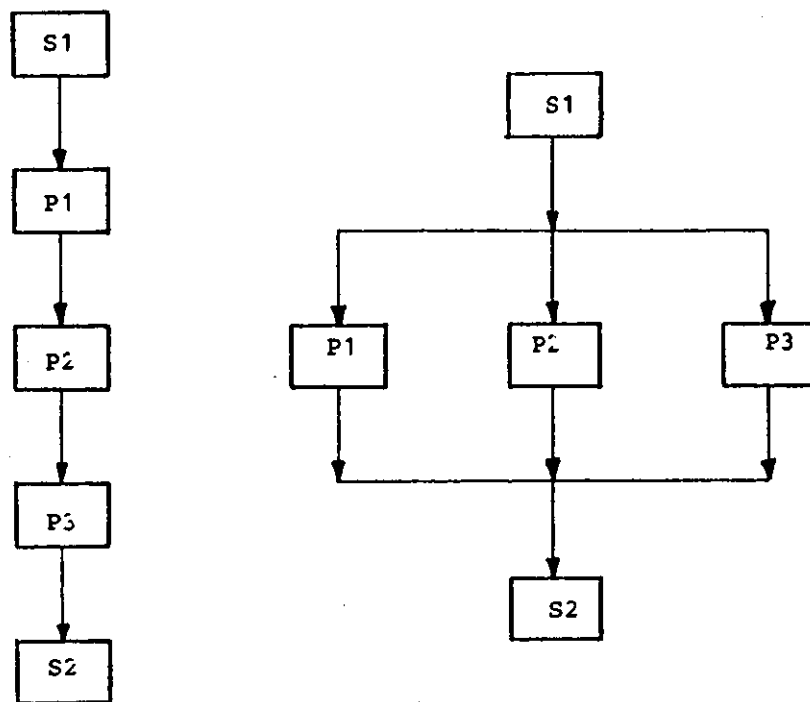


FIGURE 3.3(a): SEQUENTIAL
ALGORITHM

FIGURE 3.3(b): PARALLEL ALGORITHM

Firstly, a number of parallel paths, not necessarily equal to the number of online processors, should be created; and no matter how many processors are assigned to the job, each path should be executed by only one processor locking out all the others. This is achieved differently on the Neptune and Balance systems. In the former system, each path is executed until completion on the same

assigned processor while on the latter, the same path may involve its execution by several processors due to the dynamic re-balancing of the processors.

Secondly, data is defined in S1 and used in P1, P2 and P3 should be made available to all processors, more specifically to those executing these paths. Data defined in S1, P1, P2 and P3 should be made accessible to the processor executing the segment S2.

Thirdly all the created parallel paths should be completed before the execution of the segment S2. Another more significant issue which is not apparent in our simple example and worth mentioning at this stage is the mutual exclusion. Assuming that all the parallel paths require to access a shared variable to update it, it is necessary to include a synchronisation mechanism to prevent the shared data from being corrupted.

Thus, as we can see, there are three major factors which have to receive special consideration in a parallel algorithm, namely they are the creation and termination of parallel processes, the interprocessor communication, and the process synchronisation and mutual exclusion.

These required features which are essential in supporting parallel programming are provided either as library routines (e.g. in the Balance 8000 system) or as enhancements to the programming language (on the Neptune system).

Since all these multiprocessing systems were hurriedly implemented several decisions were made depending on what was available on site. For instance, between the two available languages, in the Neptune,

FORTTRAN and PASCAL, the former was selected as a host language for the parallel constructs. Another example is the implementation of the extensions to the FORTRAN language as introduced on the Neptune system on the newly acquired system so that most programs can be easily transferred from the Neptune to the Balance without major modifications except perhaps, for the extra restrictions imposed by the compiler. However, the Balance 8000 system provides a library of subroutines which can be used as they are or as a basis for developing customised routines, tailored to specific needs, to support parallelism in all the available languages (i.e. FORTRAN, PASCAL and C). To be more specific, the library provides routines that will initialise a shared memory, of a desired size and virtual address, initialise all the synchronisation services, dynamically allocate shared memory, provide synchronisation mechanisms (blocking, locking and unlocking), determine the number of currently configured and online processors and, finally, perform processor termination and clean-up.

As with the Neptune, the introduced pseudo-FORTRAN syntactic constructs are converted to FORTRAN calls to parallel library routines by a preprocessor program that runs before the normal FORTRAN compiler. The following sections describe the routines the Balance parallel programming library provides and also the pseudo-FORTRAN constructs.

In a common memory multiprocessing system, variables loaded in shared memory are accessible to all processes in the same way that global variables are accessible to all subroutines. The way the shared variables are allocated to the shared memory depends on the programming language; in C programs they are allocated dynamically, whilst in FORTRAN programs they are allocated statically. The

pseudo-Fortran construct to declare shared variables is:

\$SHARED variable list

This is somehow equivalent to the **COMMON** statement and has the effect of loading the listed variables into the shared memory whilst the rest of the data, including the program code, is loaded into the local memory.

The use of labelled **COMMON** statements is another alternative of declaring shared variables. In this case, the label names should also be declared in the compiling command to inform the compiler which **COMMON** blocks are shared. In fact, the **\$SHARED** construct itself, is expanded to a labelled **COMMON** statement, with a preset label name, after the preprocessor stage.

Synchronisation (or coordination) between parallel processes is a requirement that must also be taken into consideration in any MIMD multiprocessor system, otherwise shared resources cannot be prevented from being corrupted whenever more than one process accesses the same data structure simultaneously. Several synchronisation approaches that enable shared resources to be accessed in a controlled manner have been proposed. The algorithms proposed so far can be broadly classified into two groups, resource-master and bartering (although some work has been couched in terms of communicating sequential processes) [Newman 1984].

In the resource-master class of algorithms, as its name implies, the resource is always owned by one processor which after using it passes this resource to another processor wishing to become the owner (or the master). It is the current resource holder that is in

a position to allocate the resource; thus the most that can be accomplished by the others is to indicate their wish to become the resource owner.

In the case of a resource-sharing system based on bartering, the resource is usually unowned. A processor wishing to own the shared resource has to perform a bidding algorithm at the same time as any (and possibly many) other processors which also desire to use the same resource. The bidding algorithm must ensure that only one processor owns the resource.

Comparing the performance of these two approaches under different workload situations (i.e. the amount of processor interference over the resource), the authors of the above mentioned paper proposed an alternative algorithm, the hybrid approach*, that behaves either as a bartering or as a resource-master algorithm depending on the amount of resource utilisation. More specifically, if the resource is found to be already owned, then it is assumed that eventually the ownership will pass to the processor on the resource-master basis, and it can wait passively for this to happen; otherwise it has to proceed to the 'active' bartering algorithm. Through a simulation study, the hybrid algorithm showed better performance characteristics than the resource-master and the bartering algorithms taken separately.

The task synchronisation and mutual exclusion problems are tackled on the Balance 8000 system by the provision of semaphores which ensure the coordination of the multiple processes actions. The

* Currently implemented on the Neptune system

simplest of these are the lock (also known as spinlock) and counting/queuing.

To ensure exclusive access to a shared data structure by a single processor, a lock with two possible values (locked and unlocked) is utilised. A processor wishing to have exclusive access to a particular shared data structure must wait until the lock associated with that data becomes unlocked, indicating that no other processor is accessing the data. The processor then locks the lock, accesses the data structure, and unlocks the lock. While a processor is waiting for a lock to become unlocked, it spins in a tight loop, producing no effective work - hence the name of spinlock. It is impossible for two processors to acquire the same lock at the same time since the hardware locks provided on the Balance system are atomic locks (i.e. the actions performed to acquire a lock are performed as a single indivisible action).

The counting/queuing semaphores can be very useful in the case when several processors are waiting for the same lock, since it is not guaranteed that the first requesting processor would be the first to acquire the lock. The counting/queuing can also be used for managing several instances of a given resource.

Semaphores can also be used to handle events (an event is something that must be awaited for before a process can proceed) and to raise or lower barriers (a barrier is a synchronisation or rendezvous point for two or more processes) depending on the program.

As mentioned before, a critical region is a section of a parallel program code forced by the user to be executed by one processor at a time in order to safeguard the integrity of the shared data

structures manipulated within this segment. The critical region starts with a lock operation and ends with an unlock operation. Figure 3.4 shows how the lock mechanism is utilized to prevent multiple processors from executing the same critical region simultaneously. It indicates clearly what happens in the case when three processors try simultaneously to execute a critical region. All processors attempt to acquire the associated lock immediately, but only one succeeds. For our example we assume that P1 is the successful one; thus P2 and P3 must wait spinnlocking while P1 executes the protected program code. When P1 releases the lock, P2 and P3 again attempt to acquire it, and this time P2 wins, P3 must wait again.

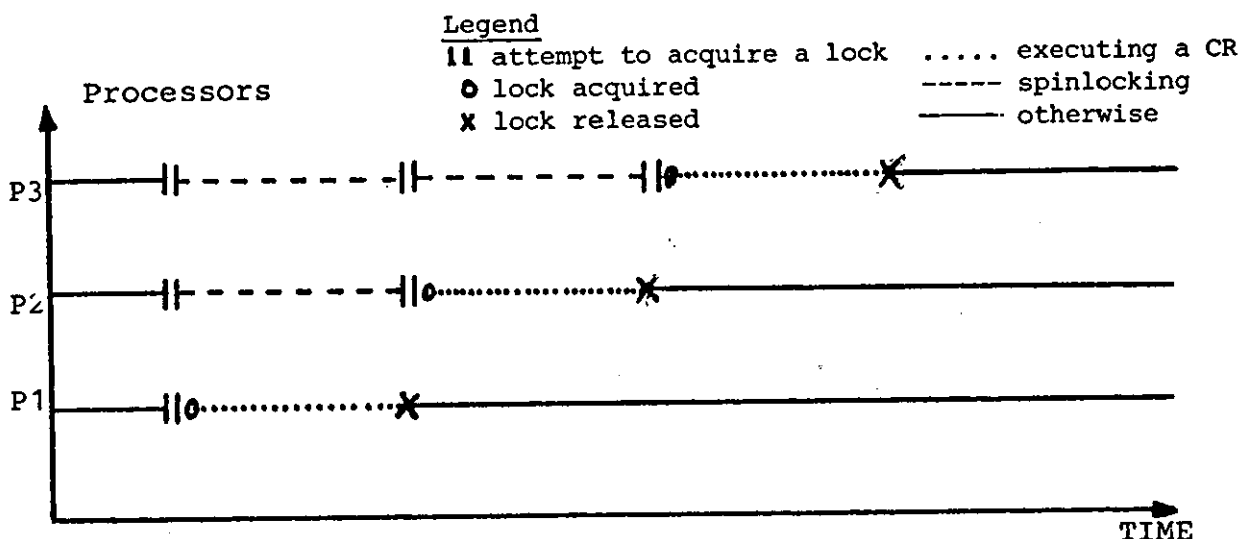


FIGURE 3.4: USE OF A LOCK IN PROTECTING A CRITICAL REGION

The system provides up to eight software resources which can be exclusively owned by only one processor at a time. These resources are declared using the pseudo-Fortran construct:

\$REGION name list

where the names in the list are FORTRAN-like names. The scope of the above statement is the \$END construct. Two constructs, namely the \$ENTER and \$EXIT have been implemented to respectively claim and release a resource. Consequently a critical region can be set as shown below

```
$ENTER  Resource 1
```

```
code
```

```
$EXIT  Resource 1
```

Actually these two constructs are converted to a lock and unlock operation respectively. The same resource can be used to protect different but not nested critical regions, which may also include subroutine calls. Although claims for resources can be nested, the user should pay considerable attention to this matter so as to avoid the possibility of deadlock situations arising.

In both multiprocessing systems, the Neptune and Balance, the FORK/JOIN pair of library routine calls are used for the purpose of creating and terminating multiple parallel processes. In fact, these two terms were significantly used in the literature to describe the process of switching from a single instruction stream to multiple instruction streams (fork) and then back again to a single stream (join). However, the Balance library routine call to a fork) procedure generates a new (child) process which is a duplicate copy of an old (parent) process, with the same data, register contents and program counter. The child process also has

access to the same opened files and shared memory space as the parent. As we have just seen, physically there is no difference between the two types of processes, however for some reason it is required to establish the parent-child relationship between the processes by returning zero (0) to the new forked process and the child's process ID to the parent. From this point the parent and child are two separate entities that can be performed separately.

Since it is relatively expensive to initiate new processes on the Balance system (about 60 milliseconds per process) it is quite normal that the number of *fork/join* calls should be kept to its lowest required value. Therefore, a parallel application typically generates as many multiple processes as it is likely to need at the beginning of the program and does not terminate any of them until the complete execution of the whole program. Thus a process which is not needed during certain code sequences, can either wait in a busy loop or relinquish the processor until it is needed. By contrast, the cost of this latter operation is between 1 to 2 milliseconds.

Three pairs of pseudo-FORTRAN constructs to generate/terminate parallel paths have been implemented. These are the \$FORK/\$JOIN, \$DOPAR/\$PAREND and \$DOALL/\$PAREND parallel constructs which are explicitly outlined below.

The first type of construct is used to generate parallel paths with different codes. A typical segment of a program using the \$FORK/\$JOIN construct might be:

```

                $FORK      label1, label2, ..... labelm; label
label1      code1
label2      code2

labelm      codem
label:      $JOIN

```

where each code i , except for code _{m} , must include, as its last instruction, a GOTO statement branching to label.

This construct is analogous to a computed FORTRAN GOTO; it generates m parallel paths, each starting at the corresponding labelled instruction. The GOTO statement included at the end of each code segment, except segment m , is used to force all the paths to terminate at the label of the \$JOIN statement.

Alternatively to the above construct, the \$DOPART/\$PAREND can generate and terminate paths with identical code. The general syntax format of such a construct is depicted below:

```

$DOPAR      label  var = exp1, exp2, exp3

              code

label      $PAREND

```

where var is an integer variable, and exp1, exp2 and exp3 are integer expressions (exp3 may be omitted if it is equal to unity).

This pair of parallel constructs is similar to the FORTRAN DO/CONTINUE statement. It creates $(\text{exp2}-\text{exp1})/\text{exp3}$ identical paths, each with a different value of the loop index, var. Thus the indexing of the loop allows different paths to evaluate different results. The number of generated paths can be as large as the highest positive integer that can be represented on the Balance (2,147,483,647).

In order to generate exactly as many identical paths as there are processors, each of which is forced to execute one and only one path the \$DOALL has been provided. The format of how to use this construct is:

```

                $DOALL      label

                code

label          $PAREND

```

This is very useful to initialise data or obtain timing information from all the activated processors. The difference between this construct and the two previous ones is that there is a unique path associated with every configured and on-line processor. The \$PAREND statement, as used in \$DOPAR and \$DOALL, is used to indicate the terminate point for all the created paths.

Despite the fact that nesting is allowed, one should notice that the local variables of an ancestor path are not available to the children paths. However under the current implementation, all the index values of the parents, together with the index value of the path, are restored prior to the execution of a path.

In addition to these parallel constructs, there are three other necessary constructs. One of these, \$USEPAR, which must be the first executable instruction of a parallel program, is used to enforce all, but one, processor to wait until more paths are created for them to execute.

The remaining two are the \$END and \$STOP which are replaced by the preprocessor program into FORTRAN statements END and STOP respectively. The \$END statement is used to force checking at pre-compile time that the nesting of parallel syntactical constructs is complete within each individual subroutine code. Whereas the FORTRAN STOP ensures the graceful termination of the program.

3.2.1 THE USER-INTERFACE TO THE NEPTUNE SYSTEM

Unlike the Balance 8000 system which provides a powerful user interface through the UNIX Operating System and its parallel extension version DYNIX, the Neptune parallel computer has seen few developments in order to enhance its user interface, the SCI - 'System Command Interpreter', mentioned in the previous chapter (see Section 2.3.3.2). Several special commands which can considerably simplify the process of implementing parallel applications on the system have been introduced. They are broadly grouped into two classes; the first class is a set of commands related to the creation of executable code (or the so-called load module) from the user's source program. The second set of commands is related to running the load module.

After the source program has been finalized and fed to the system using various SCI commands, in particular the XE- 'eXecute Editor,

the next stage can begin, involving the compiling and linking of the desired source program so as to provide the load module.

There are several commands which can be used in the process of creation, deletion and installation of load modules. The most important one used to create load modules from the user's source program is the XPFCL* - 'eXecute Parallel Fortran Compile and Link' Command which involves the following main stages:

1. pre-process the user's source program by converting all the included parallel constructs into their equivalent FORTRAN statements,
2. compile the resultant FORTRAN program,
3. link the obtained compiled code with available FORTRAN libraries and machine code written routines to control parallelism, and
4. store the created load module in the user's defined program file.

At the end of each one of the above stages, the system outputs a progress report in the form of ERROR/NO ERROR message to the user. In the case of an error occurring during any stage, the current command is terminated and an explanatory message is output to the file associated to that task stage. The file can then be viewed using the SF - 'Show a File' Command.

In Figure 3.5, we can see how a source program file, M.PARSYS.SOURCE, is 'XPFCLed' to produce the program load module named by SEARCH. The output from the pre-processing, compiling and

* This has been currently superseded by two more powerful commands, the 'XPFCLD and XPFCLX commands.

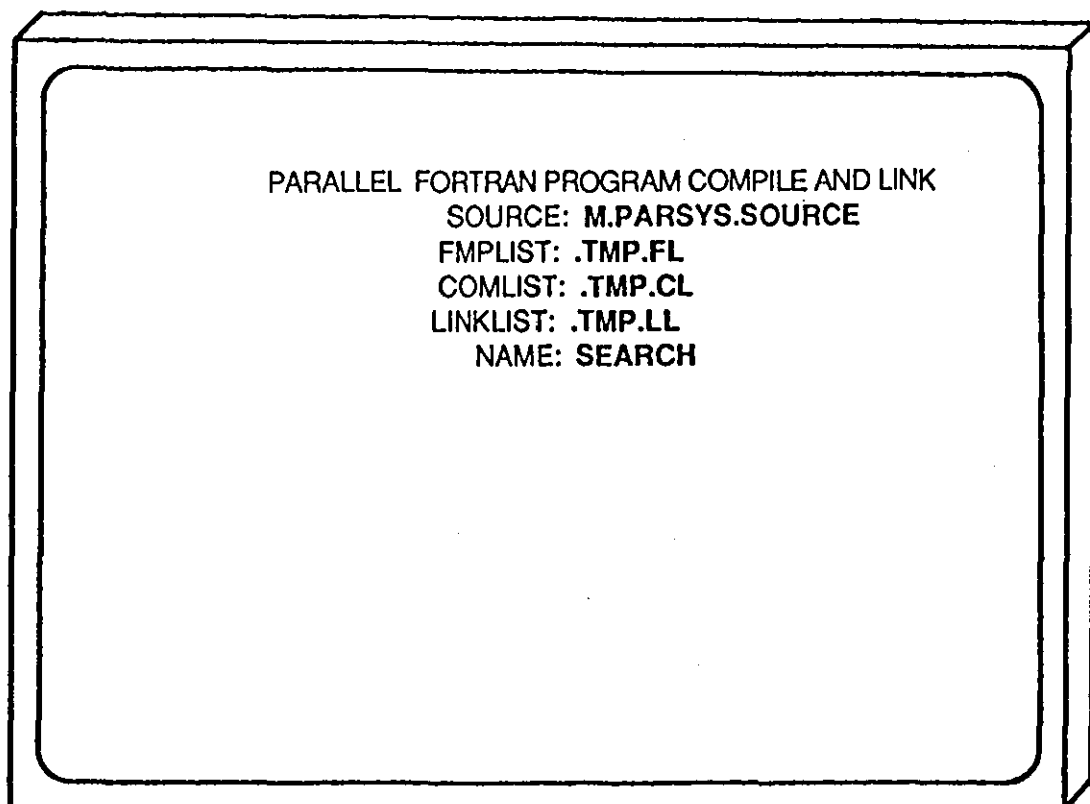


FIGURE 3.5: THE SCREEN DISPLAY OF THE XPFCL COMMAND

linking stages are written to the following files: .TMP.FL, .TMP.CL and .TMP.LL respectively.

Besides the XPFCL command, which in fact produces a parallel load module (i.e. consisting of 2 segments: a segment for the shared data and another one for the code and local data), additional commands such as the XPFCLS and XPFCLN commands have also been introduced. These commands are similar in that they both produce a sequential load module but with different characteristics; the module produced by the former command would cause the shared data to be loaded in the local memory while in the case of the latter command it would be loaded in the shared memory. As we shall see in the following section (3.3), these modules are very useful in determining various overhead timings of a parallel program.

Other commands related to the compiling/linking phase are, the DPT - 'Delete Parallel Task', the IPT - 'Install Parallel Task', the MPF - 'Map Program File', the WAIT - 'Wait for background task to complete', and the KPT - 'Kill Parallel Task' commands. For the first two commands, the system prompts <NAMES:> and waits for the load module name to be typed in. The DPT, as its name suggests, deletes a named load module before a new one is installed whether as a result of executing the XPFCL command or by using the IPT command. This deletion is necessary since it is not possible to overwrite an already existing load module. The user can use up to 256 differently named load modules.

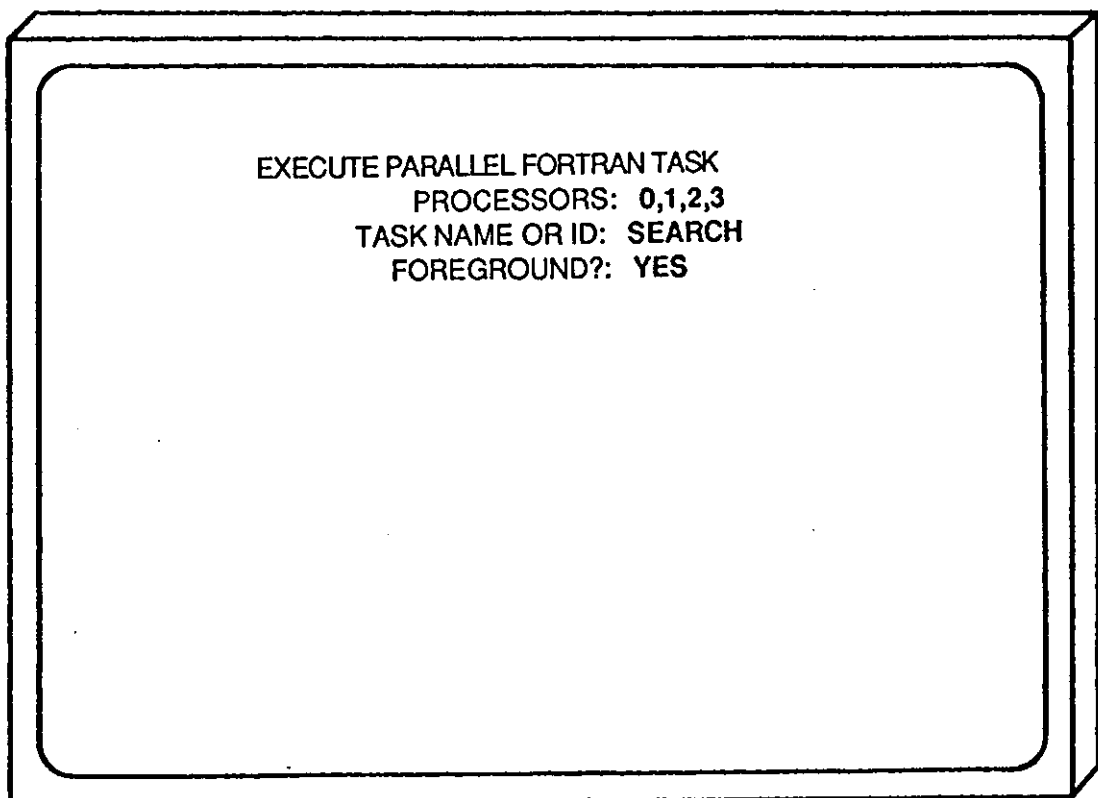
To recover from a failure in the installation phase of the compiling/linking task, due to the already existent installed name, the user can still install the load module from the linked output, by issuing the IPT command rather than repeating the whole process of compiling/linking. In the case that an installed program name has been forgotten, the list of all defined names can be viewed by using the MPF command.

The last two of the above listed commands are used to manage the parallel tasks running in the background environment; the WAIT command causes the SCI to wait until the background task is completed, producing a termination report, while the KPT command 'kills-off' this background run.

The next phase that logically follows a successful creation of a load module is the task of running the resultant module whether to debug it or to measure its run-time etc. Whatever the reasons are, the user should, prior to the actual program run, assign all the

Input/Output channels, included in the FORTRAN program, with a file or device name by using the command `AS - 'Assign Synonym'`. Alternatively, if direct Input/Output communication with the user terminal is preferred then the units 5 and 6 (used in a FORTRAN `READ` or `WRITE` statement respectively) should be assigned to the value `'ME'`. Thus a program under the above unit assignments would accept all the input from the keyboard and outputs all its results to the screen.

In order to run a load module the `XPFT - 'eXecute Parallel Fortran Program'` is used. As it can be seen in an example of an `XPFT` session shown in Figure 3.6, the user should specify the required processors, the name of the load module, and whether this execution is to be performed on a foreground or background basis environment. However a background task should not involve any input/output operation to the terminal. The state of a background can always be inspected since the `SCI` still remains operational.



```
EXECUTE PARALLEL FORTRAN TASK
PROCESSORS: 0,1,2,3
TASK NAME OR ID: SEARCH
BACKGROUND?: YES
```

FIGURE 3.6: `XPFT` COMMAND

In the Neptune system, processors are numbered 0 through 3 and any combination of them can be used to execute a parallel program, as long as the initiating processor (i.e. the one 'logged in' on) is included in the list. Errors occurring during program execution as well as terminating conditions from each involved processor are reported to the user. A listing of the commonly known run-time errors is given in [Texas Instruments, VI]. In the case of a correct program execution the system reports the following message:

STOP 0

NORMAL PROGRAM COMPLETION

from each processor. Figure 3.7 shows a typical XPFT report of a non-running program which was stopped by a 'break key'.

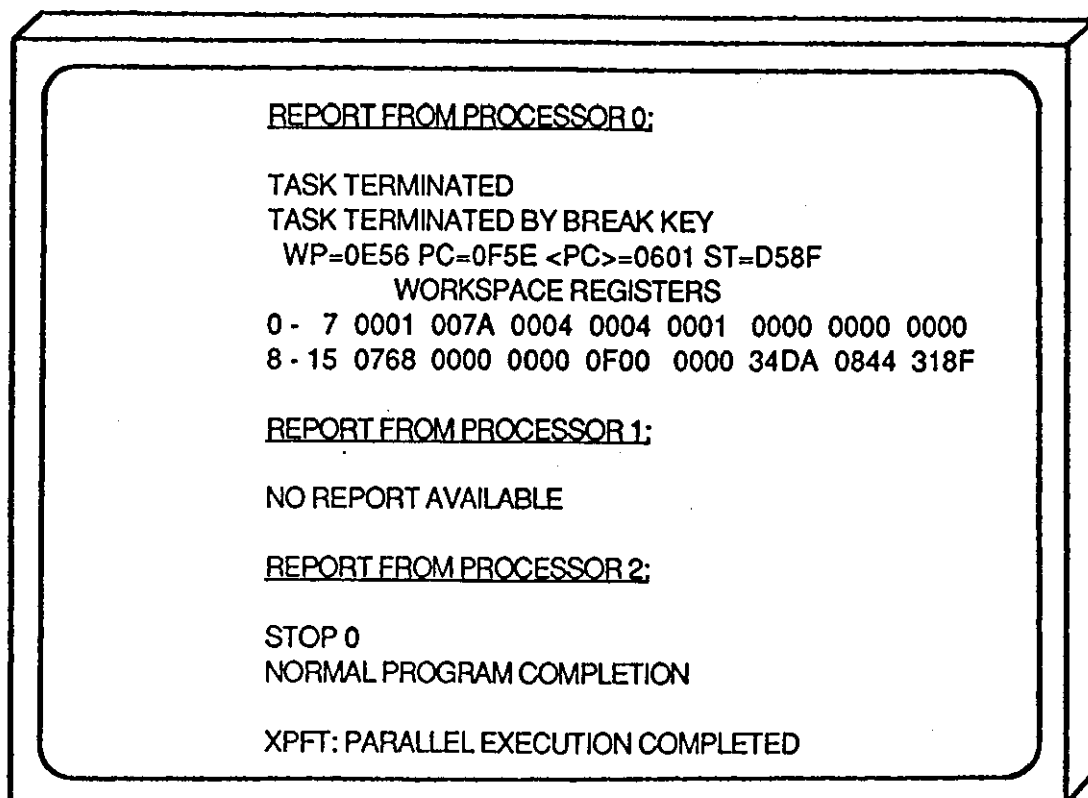


FIGURE 3.7: A TYPICAL XPFT REPORT FROM 3 PROCESSORS

Sometimes, the same XPFT is performed several times so that an average run-time behaviour can be deduced. Therefore, a more advanced execution command, the XPFR - 'Repeated' can be utilised.

Finally, the SOPR - 'Set up Overnight Parallel Run' which allows runs to be made at night rather than requiring exclusive access to the whole system during program timing, has also been provided. A user may during one log-in session, set up to ten overnight jobs, each consisting of a certain number of executions of the same program with the same data on various processor combinations. Then, the system enters every run in an XOB - 'eXecute Overnight Batch queue' which will be served on the first-in-first-out basis with a maximum allowed processing time of 120 minutes per job. The user has limited access to the XOB queue, which can only be viewed by using the XOBQ command. However, job entries which are no longer desired can still be removed from the queue using the XOBD - 'XOB Delete' command.

3.2.2 PARALLEL CONTROL SCHEME IN THE NEPTUNE SYSTEM

A scheduling procedure that implements the parallel path execution scheme for XPFCL was originally developed by Dr H Barlow. The scheduler algorithm is based on the utilisation of a shared array of up to 75 path descriptor blocks*, mutually protected through shared resource number 9. Each path descriptor block contains the following information:

- starting address of this path
- address of the parent path descriptor block

* The so-called TCB's - 'Task Control Blocks'

- variable index address (for a DOPAR construct)
- current value of the index value
- processor id, scheduled to execute this path (0 if any), and
- count of active children paths.

The routine INIT, derived from \$USEPAR construct, initialises every path descriptor block as empty and then sets up a single block describing the first sequential segment that starts the program. All the processors but one try to get a path to execute by calling the scheduler program. In the case when there are no paths to execute, the inquiring processor is forced to wait by entering an idle loop within the scheduler itself before having another go. The wait cycle time is 10 ms).

When a processor executes a FORK subroutine, known to the parallel programmer by its equivalent parallel constructs (i.e. \$DOPAR, \$DOALL or \$FORK), the array is searched for as many empty blocks as the number of paths to create. If the search is successful, then for each created task, a path descriptor block is entered in the array, otherwise the error message is reported and the calling processor enters an idle loop before re-scanning the array until sufficient empty blocks are found.

For the \$DOALL construct, a path descriptor block is created for each activated processor. As a consequence of the one-to-one relationship between processors and paths each processor id is set on one of the selected blocks.

On the other hand, the \$DOPAR creates a block for each index value of the construct variable and sets this value in the created block. Conceptually, any processor may choose any one of the DOPAR path

blocks, however, since the array is scanned in some orderly fashion, the index values are chosen in the same order as for the sequential DO-loop. When taking a DOPAR path to execute the index variable is set to the index value held on the block, therefore this variable must be in private memory otherwise several processors would try to set the same location.

Logically, \$DOPAR should be allowed to be used in any nested structure of unlimited levels. However, due to the complexity of the introduced problem, namely the parent path index variable may not hold the same value as that of the parent of the current path, the nesting levels were limited to only four. More specifically, consider an example of a program, such as the one shown below, using nested \$DOPAR construct:

```

                                $DOPAR    100      I=1,3
                                $DOPAR    200      J=1,3
                                $DOPAR    300      K=1,2
                                WRITE (6,10)  I,J,K
300                             $PAREND
200                             $PAREND
100                             $PAREND

```

Obviously, this is not an efficient parallel program since the WRITE instructions which may be issued in parallel are, in fact, performed sequentially after being queued on the spooling device. When a processor get a new path from the K-\$DOPAR (after possibly executing another one of the same type), the I and J values left from the

previous path execution might not be the same as that of the parent of the taken paths. This is a result of the fact that a processor selects a path from the available \$DOPAR created paths irrespective of the previous executed one on that processor. The solution to this problem is to get all the index variable values of the parents by backtracking the array structure and copy them in the local memory before the actual path execution. This solution has a significant disadvantage since the time to set up the parent index values for each executed path is proportional to the depth of the nested structure. Fortunately, nested \$DOPAR's are so rarely found in parallel programs that the overheads of setting up the parent values are almost negligible.

As indicated earlier, parallel processes are terminated by a CALL to the JOIN routine, obtained from pre-processing the \$SPAREND/\$JOIN constructs. When called, the JOIN routine decreases the count of active children by one, if the count is zero, indicating that there are no outstanding paths to execute, then the parent path descriptor block is released by setting it empty and the caller can (if allowed) execute any path following the JOIN construct.

Finally, all the operations that modify the contents of the shared array are made within a critical region, governed by the same resource (resource nine). The benefit gained from such an organisation is that the TIMEOUT routine will return the number of accesses and wait cycles involved in the scheduling control, as well as any other additional used resource in the program.

When the XPFCL path execution scheme was used to run parallel programs, its deficiencies were slowly identified. The problems encountered are the following:

1. The limit of 75 path descriptor blocks was found too low. Even though it has been seen that the most efficient parallel programs tend to utilise a much smaller number of paths, it is, sometimes, quite useful to be able to measure the actual performance degradation as more paths are created. Thus, very often, the limit is far exceeded;
2. The structuring of the path descriptor blocks as an array rather than a list, increased the parallel path scheduling overhead by a term quadratic in the number of created paths. This is so, since the empty blocks are unnecessarily scanned when searching for a new path to execute. A queue organisation of two separate lists would significantly reduce this effect;
3. The critical resource routines \$ENTER/\$EXIT are based on the hardware indivisible test-and-set instruction which locks out the shared memory during the execution of the critical region code. Consequently the memory would be locked for a longer time than any other instruction and this has resulted in the occurrence of several memory time-out faults. A combined resource sharing algorithm, developed in the Department of Computer Studies, has successfully eliminated these memory faults but at the expense of an increased execution time;
4. It is not necessary to utilise a block for each \$DOPAR path. If a single block is utilised to describe a set of \$DOPAR parallel paths, the block management operations would be simplified and thus reduce the parallel path scheduling overhead.

The suggested new facilities to solve the problems introduced by the XPFCL were finally implemented to give rise to a new version, called

XPFCLD. Considerable care was taken so as to ensure that none of the incorporated changes would affect the behaviour of existing parallel programs, but would obviously decrease the actual timing obtained when running parallel programs compiled with the new XPFCLD command.

In the new scheme implementation program, the path descriptor blocks are organised into two separate linked lists of up to 60 blocks, as compared to 75 in the XPFCL, a free list containing empty blocks and an active list to hold blocks describing active parallel paths. Thus the creation/termination of parallel paths basically involves 2 elementary operations, the insertion/deletion operations. The length of a path descriptor block was extended to 20 bytes, instead of 12 bytes for XPFCL, thus allowing additional information to be stored. The complete list of all the information contained in the new TCB is the following:

- pointer to the next block in the list (-1 if end of list)
- pointer to the parent block
- type of the block (DOPAR, DOALL, JOIN and empty)
- starting address of the path
- index variable value of the parent block (if DOPAR)
- the id of the processor that sets-up the parallel paths
- index variable address
- current index variable value
- index end value, and
- index increment value

where the last four parameters which are specific for a \$DOPAR construct, can be used as flags in the case of a DOALL construct.

When a CALL FORK is executed, a block is taken from the head of the empty list (sometimes the current empty, but not released, block can be re-utilised instead) and initialised accordingly to the actual parameters, the processor number, and the current index value of the parent path. The processor then enters the normal scheduling routine, scanning the list of active blocks for one with outstanding work. This will be either a DOALL with this processor yet to run it, a JOIN block to be executed or a DOPAR with outstanding paths.

In the case of a DOPAR, the next index value is computed (i.e. current value + increment value) and then set into the local memory; if this is the last path that has just been completed, the \$DOPAR block is set as an empty block and the nested \$DOPAR index values are set in the same way as in the XPFCL case.

The JOIN operation is even more complex than that of a DOPAR construct. When a path terminates, the count of active paths is decremented by one; if it is zero, and the block is an empty one then all the paths described in this block are completed and the block is turned into a JOIN block. This block is similar to a \$DOALL block in the sense that it is executed by one processor (i.e. the one that executed the original FORK).

A similar fate befalls a \$DOALL, when all the processors have completed their paths, the block is released (i.e. inserted in the empty list) only if it is of the type JOIN. It is these JOIN blocks that are simply re-utilised when a CALL FORK occurs, as the only information required for such a block is the address of the parent path and the id of the executing processor.

These rather complex rules ensure that the number of block operations (link and delink operations) are kept to the optimum possible value. Indeed, for a program without nested parallel constructs, such as the example shown below, only one scheduling block is utilised throughout the program, involving only one link operation.

```

                                $USEPAR

                                $DOALL  100

100      $PAREND

                                $DOPAR  200  IP = IS,IE

200      $PAREND

                                $DOALL  300

300      $PAREND

```

The utilisation of a single block for each DOPAR, has enabled the maximum number of parallel paths to be limited by the word length* rather than by the scheduling workspace size. Performance measurements showed that although the path block set-up time is about the same as for XPFCL, the new version has considerable low parallel path overhead.

* It is 15 bits+1 sign bit, i.e. $2^{15}-1 = 32767$ possible parallel paths per DOPAR block

Finally, to keep up with the development in the Neptune system, a new XPFCLX command has been implemented. It is very similar to the XPFCLD one, but with the significant advantage of extending the nesting levels of the XPFCL/XPFCLD commands from four to a hundred for \$FORK and unlimited nesting for \$DOPAR/\$DOALL.

3.3 PERFORMANCE CHARACTERISTICS MEASUREMENTS OF PARALLEL ALGORITHMS

The major resources on which the performance of a sequential algorithm is measured are the time required to execute the corresponding program and the memory space needed for such an execution. Each of these measurements can be analytically expressed by a customarily defined complexity (or cost) function $\phi(n)$, where n represents the size of the considered problem. Accordingly, one may speak of either the ~~time-complexity~~ or the ~~space-complexity~~ function or refer to either of them as simply a complexity function. With respect to the hardware characteristics of the computer system, used to execute the selected algorithm, the time-complexity function directly depends on a third performance measure: the computational-complexity function. Such a function can be used to find out estimates of the power required to solve a given problem, being measured by the number of arithmetic and logic operations involved.

A further clarification in general, is that in particular for the analysis of the computational-complexity of algorithms, it would be convenient to establish two further branches: the algebraic and analytic-complexity measures. The study of the former measure may answer several important problems such as:

1. the number of arithmetic operations used in a particular algorithm

2. the number of arithmetic operations required to solve a specific problem, and
3. the best way to solve a given algorithm in terms of the number of arithmetic operations.

On the other hand, the latter measure addresses the question of how much computation has to be performed before obtaining a final result with a desired degree of accuracy, and focuses on computational processes which in a certain sense never end.

Seeking an analogy to the above major resources for the parallel algorithms' performance analysis, one can immediately notice that time still remains the main resource for parallel processing. However, unlike the sequential case, the time-complexity of a parallel algorithm depends not only on the complexity of the computation, but also on the complexity of the overheads, such as those created from communication, synchronisation and data exchanges constraints.

More specifically, in this paragraph, by viewing the performance as the interaction of resources demanded by a parallel program and provided by a multiprocessor system a performance prediction framework is provided for parallel algorithms specifically implemented on such a system. The principle behind the 'demand and supply' analysis of resources is that parallel processing involves the sharing of resources whose limited availability forces processes' demands to compete against each other in order to 'own' them. As a result of this competition or contention we have the three following consequences:

1. the upper limit to the number of demands that can be satisfied degrades the maximum system or program performance,
2. the scheduling mechanism, used to solve any conflicting demands, imposes an overhead on resource utilisation even in the absence of contention, and
3. in the case that the number of resource requests far exceeding the maximum theoretical limit, some of the competing demands will have to wait for the specific resource until it becomes available.

The last two performance degrading factors are referred to respectively by the static and dynamic costs of a shared resource access.

Consequently, by analysing these system properties (i.e. resources availability and allocation algorithm) under various theoretical demand patterns and by characterising program demands, one can yield the performance of a particular algorithm on a particular item of hardware. This analysis will be exemplified by a brief analysis of the shared resources provided by the Neptune parallel processor system.

Let us consider further the two alternative overall performance measures for a parallel algorithm. In particular, the study of parallel algorithms for different types of parallel machines might reveal that an algorithm requires a particular feature of the given computer to run at maximum efficiency. Also such a study may address several important questions such as how efficient is an algorithm with respect to a particular computer and how much faster it is than the sequential version or, in fact, any other parallel algorithm that solves the same problem. In order to answer these questions

objectively, two performance measurements that reflect respectively the differing, algorithm and system designer, aspects have been defined, these are the speed-up ratio and efficiency of the algorithm.

Let T_p be the time-complexity of a parallel algorithm on a p -processor computer and T_1 the time-complexity of the same algorithm on a uniprocessor system. Then the speed-up (S_p) of the algorithm on a p -processor parallel computer over a sequential processor is defined as:

$$S_p = \frac{T_1}{T_p} \leq p$$

and the efficiency (E_p) is defined as:

$$E_p = \frac{S_p}{p} \leq 1$$

In order to achieve meaningful comparison between the performance of many different algorithms, the best parallel algorithm is compared with the best sequential one, even though the two algorithms might be quite different, however they should solve the same problem.

Stone [Stone 1973] introduced some typical speed-up ratios and indicated that the best speed-up ratio is linear in p , where p is the number of simultaneously active processors. Such a speed-up is achievable with some problems that exhibit a natural iterative structure; for example systems of linear equations and many other vector and matrix applications. In some cases, problems have speed-up ratios of $P/\log P$, where \log is a logarithm to the base of 2. These performance results are less desirable, however such

algorithms are still well suitable for parallel processing. On the other hand, algorithms with speed-up ratio of $\log P$, exhibit very little speed increase when the number of processors is doubled. Such an algorithm is not suitable for parallel processing and it may be better to run it on a serial computer or on a computer with less parallelism.

In the following sections, we shall discuss, more analytically, some of the inherent limitations on the performance of a p -processor computer system due to some types of overheads associated with the multiprocessor execution but not with that on a uniprocessor system.

(Obviously, it is apparent that a multiprocessing system with p identical processors cannot complete a parallel program more than p times faster than a single processor. Therefore the speed-up factor of a parallel algorithm performed on a particular parallel system is limited by the number and power of the processing elements. It is also limited by other factors introduced by the communication, synchronisation and data exchange amongst all the processors. Also the fact that a job is subdivided into p (or possibly more) individual subtasks causes an additional overhead explicitly associated with a multiprocessor system. This task partitioning may give rise to three possible types of overheads:

1. In the event of less than p subtasks remaining to be executed while there are p processors available, then some of these processors must be kept idle until the completion of all the subtasks. This idle time can be estimated (if the processors' speed is known) by analysing the computational-complexity and the number of subtasks

2. When a subtask generates results required as input to another subtask a mechanism is necessary to ensure the proper sequencing of subtasks. This creates an overhead known as **organisational overhead**. Thus, in this case, the latter subtask has to wait until the former subtask produces the results
3. In the case that the shared database is simultaneously accessible by a lesser number of processors than the system comprises, then an overhead is incurred. Such an overhead is associated with checking the number of simultaneous accesses not to exceed the fixed limit. Consequently, the processor's time is wasted while waiting to gain access.

Summarising, two types of overheads can be distinguished, those due to the design of hardware and software and those due to the interference between two or more subtasks running on different processors, causing one or more of them to wait. The former one includes the overheads from the subdivision of the task, allocation of the subtasks to processors, contention control by hardware and software. These are all called **static overheads** since, once the number of processors, the methods of communication, synchronisation and task allocation, for the algorithm to be processed are decided, then the number of subtasks, synchronisation and shared data accesses are all properties of the algorithm itself. The second source of overheads concerns the so-called **dynamic overheads** which depend not only on the algorithm, but also on the detailed timing considerations which may vary even if the same task is executed on the same piece of hardware on consecutive runs.

From the measurements point of view, the static overheads can be determined. In particular, if subtasks are created and allocated at run-time, then the static cost is obtained by multiplying the total

number of subtasks by the cost of executing the appropriate instructions on a single processor. Similarly, by knowing the cost of one access or synchronisation, the relative overhead is estimated. Unfortunately, the same reasoning cannot be applied to the dynamic overheads. Instead, a statistical estimate can be made depending on the occurrence and duration of events (subtask creation, resource demands, synchronisation).

Finally, and despite the requirement of an exact pattern for the shared resource demands to determine in detail the waiting times, it is possible to estimate the bounds on the maximum number of processors, that can be utilised efficiently on an algorithm, from the average utilisation figures for each resource and for each task. Thus, since different hardware and software give rise to different static overheads, therefore, by knowing the specific costs associated with a particular multiprocessing system, parallel algorithms may be accordingly designed so as to minimise these costs.

In the remainder of this paragraph, we shall discuss the actual overheads observed on the two multiprocessor systems, the NEPTUNE and BALANCE 8000, available at Loughborough University of Technology. The ability of these systems to provide resources statically (i.e. when requests for resources do not contend with each other) were measured by the software supporting the parallel systems. These measurements are reported in the table overleaf (Table 3.1).

From these initial performance measurements, we notice that the Balance 8000 is a considerably faster system than the Neptune parallel computer. Comparing the average time to execute an

Computer system Actual measurements	NEPTUNE	BALANCE
<u>Memory access time</u>		
* Local	0.95	0.73
* Shared	1.69	0.93
<u>Overhead timings (no contention)</u>		
* Enter & exit a critical region	800.0	11.9
* create a set of parallel paths	1300.0	290.0
* set up and terminate a path	900.0	225.6
* waiting for a critical resource	1080.0	100.0
* waiting for a path to execute	10800.0	1000.0
<u>Fortran integer instruction timings</u>		
* Do loop cycle overhead	12.9	4.0
* Assignment (i=j)	5.7	1.5
* Addition (i=j+k)	14.9	2.8
* Substraction (i=j-k)	14.9	2.2
* Multiplication (i=j*k)	16.10	10.3
* Division (i=j/k)	20.9	14.8
* Test (IF(i.eq.j))	8.0	3.8
<u>Fortran real intruction timings</u>		
* Assignment (i=j)	13.0	3.3
* Addition (i=j+k)	643.0	11.7
* Substraction (i=j-k)	658.0	11.7
* Multiplication (i=j*k)	1111.0	13.7
* Division (i=j/k)	625.0	11.5
* Test (IF(i.eq.j))	580.0	7.0

TABLE 3.1: ACTUAL PARALLEL SYSTEM PERFORMANCE MEASUREMENTS

arithmetic operation between the two systems, it is found that integer arithmetic is about 10 times faster and floating point arithmetic is approximately 100 times faster.

In general terms, when measuring a program performance by analysing the source code a programmer, in the case of the Balance system, is required to know exactly what object code will be produced by the compiler in each separate case. Since the system performs some compile time optimizations which cannot be avoided or switched off, some performance results may not appear reasonable. Secondly, the use of cache memory on each processor has a marked effect on timings since it is difficult to predict when the cache memory will be used efficiently (a classic example of this situation is when a program loop demands more space than the cache - it then runs far slower than a very slightly shorter program).

For both parallel processor systems, specifically for the performance measurement of the three parallel resources aspects mentioned earlier, two subroutines are available for obtaining timing information. The routines should be embedded within a \$DOALL/\$PAREND pair of parallel constructs to force each processor to execute them. Thus, to start or restart timing we should perform:

```
CALL TIMEST
```

and to obtain the current timing information we use:

```
CALL TIMOUT (ITIME)
```

where ITIME must be declared as a shared array of size 144 rather than one hundred on the Neptune (to cope with the increased number

of processors). The timing block information, for each involved processor, returned by the TIMEOUT routine has the following format:

- 1 Clocked time (seconds)
 - 2 Clocked time (milliseconds)
 - 3 Elapsed time (seconds)
 - 4 Elapsed time (milliseconds)
 - 5 Total number of executed paths by this processor
 - 6 Number of cycles waiting for a path to execute
 - 7 Number of accesses to the system scheduler resource
 - 8 Number of wait cycles for system scheduler resource
 - 9 Number of accesses to user resource (critical section) number 1
 - 10 Number of wait cycles to user resource number 1
- Repeat points 9 and 10 for user resources 2 to 8
- 23 Number of accesses to resource number 9
 - 24 Number of wait cycles to resource number 9.

Such an important information, if used in conjunction with the actual system performance measurements as given in Table 3.1, should enable the user to obtain valuable estimates about the cost of each of the three necessary operations required for parallel processing (i.e. parallel control, data communication and process synchronisation). For example, the static cost of parallel control can be estimated as the product of the number of executed paths and the cost of scheduling. Similarly, the idle processor time which represents the time spent by a processor while waiting for paths to be created, can also be estimated as the number of wait cycles by the elementary waiting time.

On the other hand, the creation of parallel paths and their allocation to processors is a dynamic process which is achieved only

through a shared list mutually protected by resource number 9. Table 3.1 shows the average time this resource is blocked to other processors, while the dynamic loss of performance due to this resource contention can be estimated

From the data communication point of view, the shared data static cost for each parallel system, arising out of the hardware multiplexing of more than one processor into a single memory block, is also given. The static overhead to access shared memory, which is widely recognised to be a function of the number of contending processors and the temporal pattern of access, is increased by the contention level. However, these degradation costs are significantly smaller than those arising from mutual exclusion since accesses to the shared memory are such that they are normally more regular than anticipated.

The routines ensuring mutual exclusion to shared data structures, count the number of times each processor accesses each distinct user defined resource; consequently, the static cost of mutual exclusion is the product of that number with the unit cost of the mutual exclusion mechanism given in Table 3.1. Also, these routines can estimate the waiting time, due to the contention for each one of these resources from each processor.

In conclusion, we must clarify a significant factor contributing to the performance, related to the limited availability of the shared resources on a multiprocessor system. In particular, if the resource availability equals the total demand rate, then saturation has occurred and no more speed-up can be achieved through utilization of more processors. In other terms, this means that the maximum number of processors, which can be effectively utilised in a

parallel program, is limited independently by each shared resource according to:

$$\text{Maximum processors} = 1/(\text{demand rate} * \text{unit access time})$$

Therefore, the mean demand rate to a resource is an important measure of the best overall performance achievable, since it can also determine, together with the access mechanism properties of the system, any losses in the performance arising from processes sharing resources.

3.4 GENERAL PARALLEL PROGRAM STRUCTURING CONCEPTS

The study of parallel algorithms which is widely covered in the literature, has been found to be a fascinating and challenging research topic by many computer scientists. In fact, parallel algorithms have been studied since the early 1960's, even though there were no parallel computers built at that time. Gradually interest in these algorithms has increased by the emergence of large-scale parallel computers such as the DAP, ILLIAC, CRAY and CYBER. Since then a large variety of algorithms have been designed based on different points of view and for various different parallel computer architectures.

In the design of parallel algorithms, the particular characteristic features of the computer on which the algorithm is to be implemented should be thoroughly considered. It is obvious that different types of computer architectures, such as those described in the two previous chapters, execute different types of parallel algorithms. However it is sometimes possible for an algorithm to be implemented

on more than one type of computer by applying appropriate modifications to the algorithm.

In Stone [1973] some of the problem issues as related to parallel algorithms are highlighted. These include the requirement of data management 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 could lead to very efficient parallel algorithms and lastly the possibility of transforming a given sequential algorithm to yield new algorithms suitable for parallel processing.

Kung [Kung 1980] proposed a conceptual taxonomy for parallel algorithms based on three orthogonal dimensions of the space of parallel algorithms: **concurrency control**, **module granularity** and **communication geometry**. The requirement for concurrency control in parallel algorithms is to ensure the correctness of the concurrent execution since more than one process may be executing at a given time. The module granularity measures the maximum amount of computation that a typical process can perform before having to communicate with other processes. This can also be used to reflect whether or not a parallel algorithm tends to be communication intensive. In particular, if the module granularity is very small, the processes spend most of their time receiving and sending data. The communication geometry represents the communication network connecting the different processes in a more efficient manner.

In the following sections, we shall present a brief discussion of parallel algorithm design techniques for several parallel computers as discussed in Chapters 1 and 2, focussing more on those techniques

specially formulated for the design of efficient parallel algorithms for MIMD and SIMD computers. Many illustrative examples are also provided in order to emphasize the main characteristics of the used technique.

For the dataflow systems which have been the subject of a great deal of research activity since the late 1960's, programming is organised so that the control sequencing of the statements for execution is governed by the availability of the operand values. More schematically, machine instructions are linked into a network so that the result of each executed instruction is automatically fed into appropriate inputs for other instructions. Considerable parallelism is possible with the dataflow approach. Since no side effects can occur as a result of instruction execution, many statements can be active simultaneously. Furthermore, if the underlying architecture can support this principle, all program statements whose input values have been previously computed can be executed concurrently.

Although dataflow concepts are very attractive for providing such a highly parallel processing model, to date there have been few dataflow machines built. This does not mean that this approach is a failure but on the contrary, it shows that more efforts are still required in dealing with the main problems that confront the dataflow architectures. For instance, to fully exploit the concurrency it is desirable to design new languages that could modify a simple program and convert it almost into a form suitable for dataflow processing. It might also be useful to develop new algorithms that take advantage of concurrent computation.

As an example, let us consider the execution of the program fragment $(a = (b+c)*(b-c))$ on a typical data flow computer. The exact sequence of this execution is illustrated in Figure 3.8 where a block dot on an arc indicates the presence of a data token for the corresponding node. Assume that during some computation stages, the values of the variables b and c , indicated by two black dots in Figure 3.8a, have been generated. Since each data token is required by two different nodes, the next step (see Figure 3.8b) corresponds to the duplication of each generated token. As a result of the availability of the two pairs of input tokens, both the addition and subtraction nodes are enabled to fire (or execute). The concurrent execution of these two nodes means that each node consumes its input tokens, performs the specified operation and then releases the resultant token onto the output arc. Finally, the multiplication node is enabled, as indicated in Figure 3.8c, and its subsequent execution produces the result token corresponding to the value of a (see Figure 3.8d).

In a pipeline computer, a sequence of identical operations is 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 sequence, the greater the efficiency is. For these reasons a good pipeline algorithm is, generally speaking, a good SIMD algorithm and vice versa.

One of the most striking architectural features of the pipeline hardware has been its successful ability of handling arithmetic operations. Papers, such as those presented by Chen [Chen 1975], and Ramamoorthy and Li [Ramamoorthy 1977] discussed many pipeline algorithms for floating-point addition, multiplication, division and square-root calculations. For these algorithms, the various stages

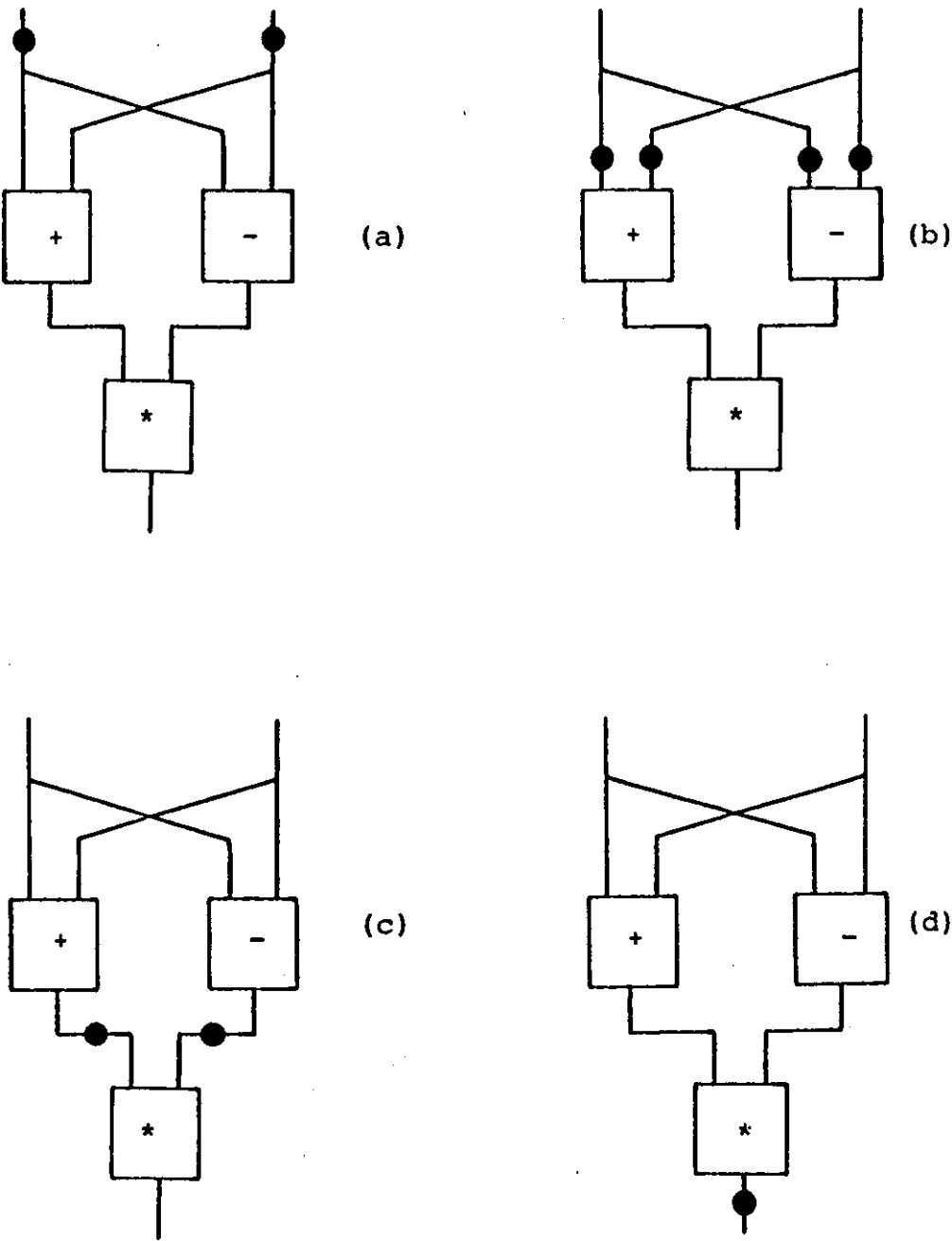


FIGURE 3.8: A DATAFLOW EXECUTION OF THE EXPRESSION $a = (b+c) * (b-c)$

of the pipe are linearly connected, although additional feedback links may sometimes be present. For instance, the CRAY-1 uses six-stage floating-point adders and seven-stage floating-point multipliers, and the CDC STAR-100 uses four-stage floating-point adders. For a pipeline floating-point adder, the pipe typically consists of stages for performing exponent alignment, fraction shift, fraction addition, and normalization. A pipeline arithmetic unit can be viewed as a set of linearly connected processors, each of which is capable of performing a specific operation.

One of the ideal situations where the pipeline approach could be most efficient is when the same sequence of operations is invoked very frequently so that the start-up time to initialise and fill the pipe becomes relatively negligible. This is the case when the machine is processing long vectors. Thus one of the major concerns in using the pipeline computers such as the CRAY-1 and the STAR-100 is the average length of the vectors to be processed. For integer arithmetic, bits in the input operands and carries generated by addition are often pipelined.

As an example, let us follow the example of a pipeline digit adder using a linear array which is described in [Chen 1975]. Suppose that we are required to add two integer vectors (U_i) and (V_i) and that each element is a k -digit integer (i.e. $U_i = u_{i1} u_{i2} \dots u_{ik}$ and $V_i = v_{i1} v_{i2} \dots v_{ik}$). In Figure 3.9, where the u_{ij} and v_{ij} flow towards the processing units synchronously, we illustrate how the pipeline digit adder works for $k=4$.

At each cycle, each processor sums the three digits arriving from the three input lines and then outputs the sum and the carry at the output lines. It is easy to check, from the figure shown below,

that when the pair (u_{ij}, v_{ij}) reaches a processor, the carry needed to produce the correct j th digit in the result of $U_i + V_i$, will also reach the same processor. Consequently, the pipelined adder is able to compute one element sum of $U_i + V_i$ every cycle in the steady state.

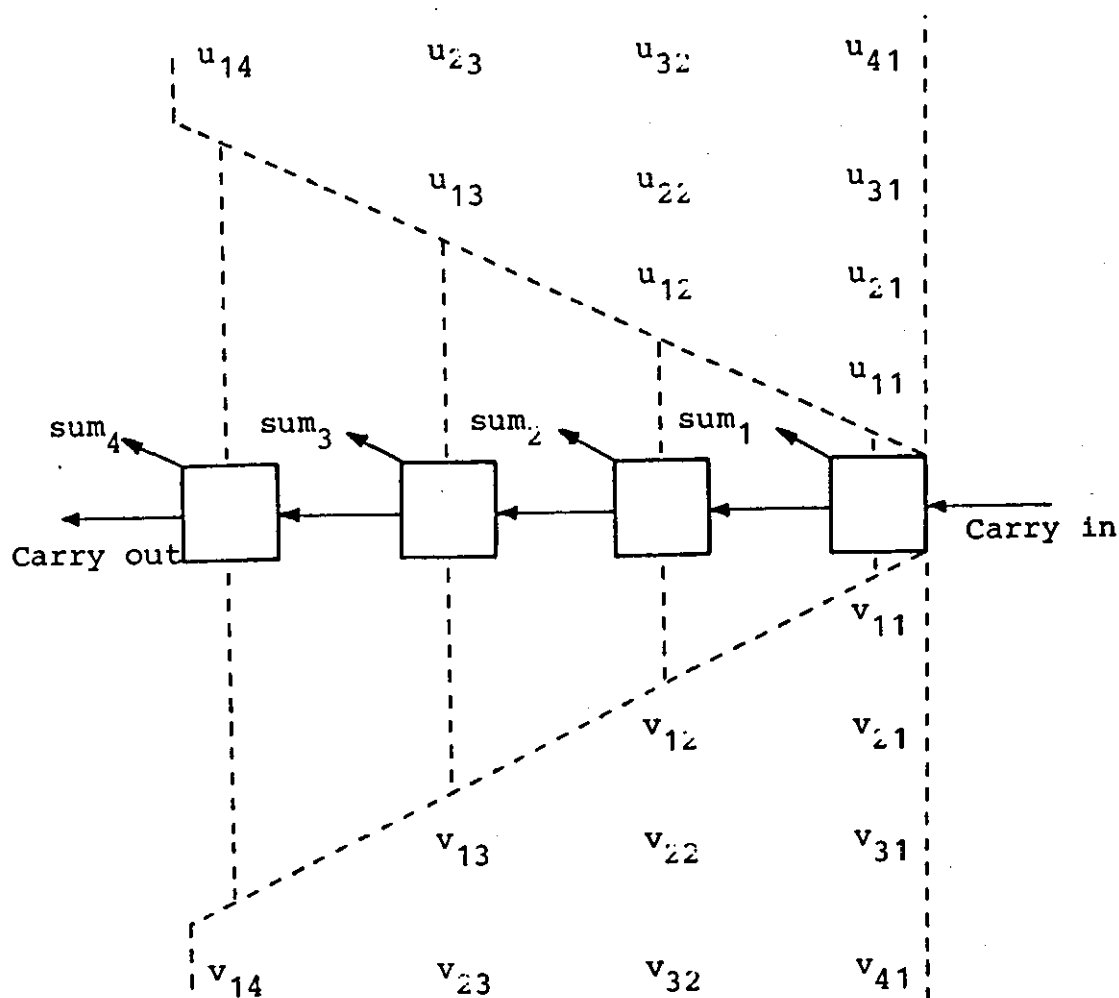


FIGURE 3.9: A PIPELINE DIGIT ADDER

As mentioned earlier, a good pipeline algorithm is also a good SIMD algorithm, and vice versa. For the SIMD parallel computers a wide selection of algorithms was designed and studied in the literature (see for example the papers [Miranker 1971], [Stone 1971], [Heller 1978] and [Wyllie 1979]). The latter reference covers mainly non-numerical problems applied to various data structures, such as the counting of the number of elements in a linked list and the insertion and deletion of element(s) from a given linked list. Another widely investigated non-numerical problem is sorting. In particular, Baudet and Stevenson [Baudet 1978] considered the implementation of sorting algorithms on SIMD computers using a generalised odd-even transposition. Nassimi and Sahni [Nassimi 1979] also presented an $O(n)$ algorithm to sort n^2 elements on an $n \times n$ mesh-connected parallel computer and Thomas and Kung [Thomas 1977] developed a sorting algorithm to sort n^2 elements on an $n \times n$ mesh-connected processor array that requires only $O(n)$ routing and comparison steps.

Let us consider, once again, the problem of adding two n -vectors whose solution was previously developed for a pipeline digit adder. This algorithm is also suitable for an SIMD computer without considering the binary representation of the elements. Thus, it is clear that a computer with n processors takes exactly one step to compute the vector sum of $(U_i) + (V_i)$, where each element is evaluated on a processor. The algorithm is extendable to the addition of two $(m \times n)$ matrices A and B to produce the matrix C where:

$$c_{ij} = a_{ij} + b_{ij} \quad \begin{array}{l} \text{for } i = 1, 2, \dots, n \\ \text{for } j = 1, 2, \dots, m. \end{array}$$

A computer with $m \times n$ synchronous processors would surely compute this sum in one step.

Several powerful methods for designing parallel algorithms for SIMD computers were suggested in the literature. For instance, Tang and Lee [Tang 1984] designed many algorithms based upon the **divide-and-conquer** strategy which is based on partitioning a given problem into a certain number of subproblems of less complexity than the original one and only when all these subproblems are individually solved, perhaps in parallel, then their partial solutions are combined into the final solution of the initial problem. As an example, consider the problem of finding the maximum of n numbers. We first partition our initial set $S(n)$ into (let us say) k subsets, each containing $\lfloor n/k \rfloor$ elements, so that we end up with k subproblems. The solution of each subproblem S_i yields the maximum M_i , $i = 1, 2, \dots, k$. The final step is a merging step, where the maximum M is selected out of the k 'submaximums'.

Another powerful method, based on problem decomposition and used to generate parallel algorithms for SIMD computers is the so called **recursive-doubling** strategy. Basically, such an approach consists of splitting up the original computation into independent smaller computations of equal complexity which can be processed in parallel on separate processors. As an example, consider the problem given below:

$$A_n = a_1 \circ a_2 \circ a_3 \circ \dots \circ a_n,$$

where \circ is an associative operation. In Figure 3.10 we illustrate how this method works on this particular example. At each level the operations are identical and independent, therefore they can be

executed simultaneously. It is obvious that a system with $\lceil \frac{n}{2} \rceil$ synchronous processors would perform this sum in $\lceil \log_2 n \rceil$ steps.

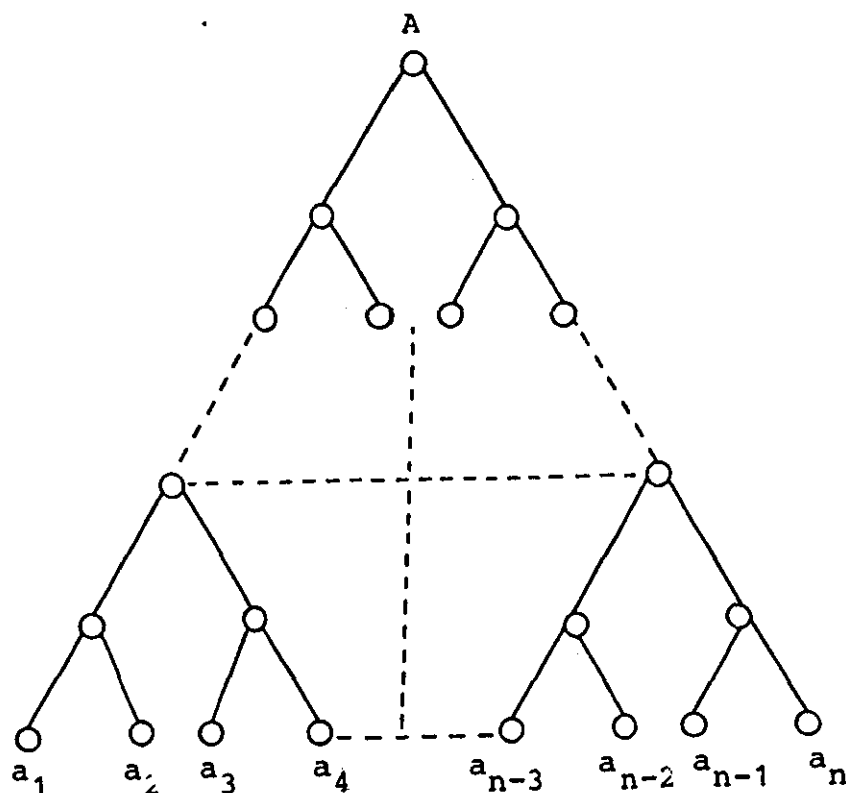


FIGURE 3.10: ASSOCIATIVE FAN-IN METHOD TO EVALUATE EXPRESSION A

This algorithm is also known as the associative fan-in algorithm [Heller 1978] and also under the names log-sum and log-product algorithms with the operators $+$ and $*$ respectively. Besides the simplicity of the associative fan-in algorithms, it was shown that they are optimal in the sense that they achieve minimal computation time for any number of processors used.

For the past two decades, parallel algorithms for the matrix manipulations on SIMD computers have received considerable research interest. Muraoka and Kuck [Muraoka 1973] investigated the valuation of a conformable sequence of matrix products A_1, A_2, \dots ,

A_n , where the dimensions of A_1 are either $1 \times N$, $N \times N$ or $N \times 1$ using unlimited parallelism power. Hockney and Jesshope [Hockney 1981] suggested three ways of matrix multiplication. The first method, IPM - 'The Inner Product Method', which is an extension of the inner product algorithm, requires $n^2 (\lceil \log_2 n \rceil + 1)$ steps using n processors since it consists of n^2 inner-products. The second method, MPM - 'the Middle Product Method', computes the inner-product over all the elements of a column of C in parallel. Using n processors, this method completes in $2n^2$ steps. The third method, OPM - 'the Outer Product Method' which computes the inner-product over all the elements of the array result in parallel requires $2n$ steps using n^2 processors. Furthermore Jesshope and Craigie [Jesshope 1980] showed that the product of two matrices can be achieved in $\lceil \log_2 n \rceil + 1$ steps using n^3 processors.

Another parallel algorithm for the evaluation of arbitrary matrix expressions is discussed in the papers by Maruyama [Maruyama 1973] and Kuck and Maruyama [Kuck 1975] where the parallelism is assumed unlimited.

Up to this point, we have been considering almost exclusively the case when there are enough processors for the problem. However, the reality is that problems are, in general, larger than the potential parallelism of the computer. Therefore, the original algorithm should be restructured so that the processing requirements of the new algorithm are reduced to a realistic figure. The efficiency of the new algorithm should be similar to using a theoretical large number of processors. Two basic approaches have been suggested by Hyafil and Kung [Hyafil 1974] so that a large problem can be solved on a realistic number of processors. The first principle, the problem decomposition, is based on partitioning a problem into

subproblems small enough to be solved on the provided number of processors. For example, a matrix multiplication involving large matrices, can be performed on a computer with a small number of processors by computing a sequence of matrix multiplications involving submatrices. On the other hand, the second method which is the algorithm decomposition technique, forces simultaneous operations involved in one step of the original algorithm to be carried out in a number of steps on the limited computer. For example, a problem requiring one step on an n -processor SIMD computer, can be solved in $\lceil n/p \rceil$ steps where p is the number of available processors.

For the asynchronous multiprocessor like the Neptune or the Sequent Balance, which is composed of a number of independent processors sharing a global memory via a shared common bus, algorithms may be viewed as a collection of cooperating processes that may be executed simultaneously in solving a given problem. Due to the unpredictable behaviour of the asynchronous processors, serious issues regarding the correctness and efficiency of an algorithm are considered. The correctness issue arises because of the unpredictable handling of the shared data by the concurrent processes. On the other hand, the efficiency issue arises because any synchronisation introduced for correctness reasons involves additional processing time and also reduces concurrency.

One of the techniques used to design parallel algorithms for MIMD computers has been the partitioning of a problem into many processes that can be executed in parallel. This task might not seem a significant one for a small number of processors, say two to four, however for several processors, say 16 to 32, or more, the problem becomes extremely difficult. Furthermore two types of problem

decomposition were described by Hwang and Briggs [Hwang 1984]; these are the static and dynamic decomposition strategies. In the case of static decomposition, the set of processes as well as any precedence relationship amongst them are known before execution. In this method, the amount of data communication is kept very low, provided the number of processes is small. Whilst in dynamic decomposition, as its name suggests, the set of processes changes during execution. Although in such a method the data exchange rate among the processes is extremely high, it can be adapted effectively to variations in the execution time of the process graph.

Parallel algorithms for multiprocessor systems were classified into synchronised and asynchronous algorithms, aimed mainly at distinguishing the algorithm with respect to the system's particular characteristic features. In the former class of algorithms, processes are forced to wait for the required inputs, while in the latter case they are allowed to continue asynchronously.

One of the classical problems which has received a considerable research interest is the root-searching problem. The definition of such a problem is that 'given a continuous (or discrete) function f , having opposite signs at the endpoints of the uncertainty interval (also called the root interval) of length ℓ , locate a zero within a unit interval'. Sequentially, this problem is approached basically by constructing a nested sequence of approximations to the root using a new point in the current root interval and computing the function value at this point. The incorporation of this newly computed point and its function value to form one of the new endpoints has the effect of systematically reducing the interval while maintaining the function values at the newly defined interval endpoints of opposite signs. Probably the best known algorithm to

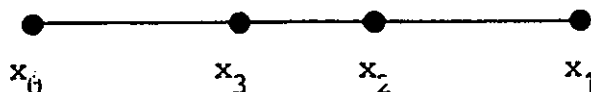
compute a new point inside the root interval is to take the midpoint of the interval; this method is known as bisection, or more widely, as binary search.

An obvious extension of the binary search method is a search algorithm using p processors which divide the current root interval into $p+1$ subintervals of equal length and evaluates simultaneously the function at each of the p division points. The parallel function evaluation is considered as one stage of the root computation process. The other stage which involves a single processor to compute a new root interval, is invoked only when all p parallel evaluations are complete. Thus, this is a synchronised parallel root-searching algorithm. It is obvious that every iteration reduces the root interval by a factor of $p+1$, and it has been shown that the order of convergence of the iteration method equals at least the number p of parallel function evaluations.

The major drawback of the synchronised algorithm is that when the times of the p parallel function evaluations differ substantially, the algorithm can be very inefficient. An asynchronous version of the zero-searching algorithm is obtained by removing the requirement that the computation process does not proceed until all p simultaneous function evaluations are complete. In the following, we shall outline an asynchronous algorithm on the line of Kung [Kung 1975], whose paper is considered a major contribution to the development of the concept of asynchronous computations.

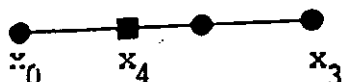
Kung, at first, introduced an asynchronous algorithm (called AZ_2) with two processors, in which the selection of the new point is based on the Fibonacci rule; later he generalised the algorithm for three or more processors.

Suppose that initially the root interval is divided into three intervals:

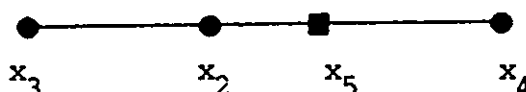


The function evaluation at x_2 and x_3 is started simultaneously by two processors. Suppose now that, without loss of generality, the evaluation at the left point, x_3 , finishes first. The assumption here is that the outcome is non-zero; for otherwise a zero is found and we are finished. Next a comparison of the value signs at the left endpoint and x_3 is executed and the new root interval derived, either as x_0 — x_3 or as x_3 — x_2 — x_1 , depending on the sign of the outcome.

If the first case occurs, then a new evaluation is carried out at the point x_4 which is defined by:

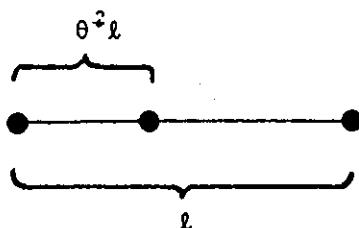


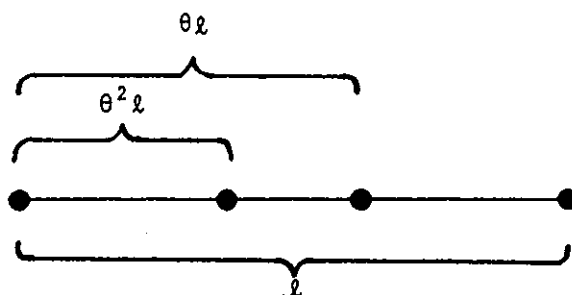
If the second case occurs, the new evaluation is carried out at the point defined by:



In general, in the process of computation one of the following states can occur, which are denoted by State 1(.) and State 2(.) and are illustrated in the following graphs:

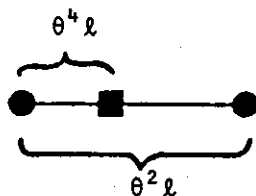
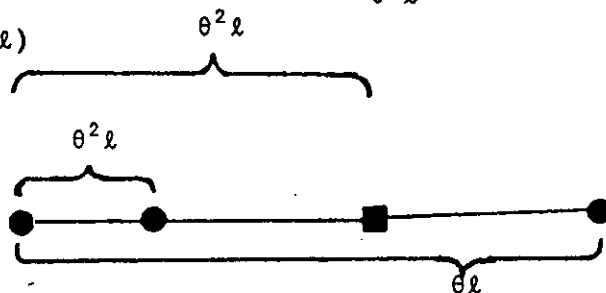
State 1 (l)



State 2 (ℓ)

where $\theta^2 + \theta = 1$, i.e. $\theta = 0.618$ is the reciprocal of the golden ratio, $5/13 + 8/13 = 1$, $8/21 + 13/21 = 1$, $13/34 + 21/34 = 1$, etc.

State 1 (ℓ) is the state for which the root interval is of length ℓ and the function is evaluated simultaneously at the point '•' inside the interval and another point outside the interval (not shown on the graph). Similarly, State 2 (ℓ) is the state for which the root interval is of length ℓ and the function is evaluated simultaneously at two points, both inside the interval. We further deduce that State 2 (ℓ) is transmitted after each computation to either:

State 1 ($\theta^2\ell$)State 2 ($\theta\ell$)

This transition is denoted by:

$$\text{State 2 } (\ell) \rightarrow (\text{State 1 } (\theta^2\ell) \vee \text{State 2 } (\theta\ell))$$

The corresponding rule for State 1 (ℓ) is

$$\text{State 1 } (\ell) \rightarrow (\text{State 1 } (\theta^2 \ell) \vee \text{State 1 } (\theta \ell) \vee \text{State 2 } (\ell))$$

These transition rules completely define the asynchronous parallel algorithm. Suppose that the algorithm starts from State 2 (1). Then assuming that the function does not vanish at any of the evaluation points, the progress of computation can be represented as a transition tree with nodes representing stages. The algorithm follows one particular path of the tree, depending upon the input function and the relative computation speed of the two processors.

For the analysis of the root-finding problem with p processors, we have noted earlier that every iteration reduces the length of the root interval by a factor of $p+1$. Hence, the algorithm requires $\lceil \log_{p+1} \ell \rceil$ iterations to find the root. Letting the time required to evaluate the function at a point in the root interval be a random variable with mean \bar{t} , the time-complexity of the synchronised algorithm can be shown to be $\lceil \log_{p+1} \ell \rceil \lambda_p \bar{t}$, where λ_p is the penalty factor for synchronising p function evaluations. For $p=2$, the expected time-complexity for the synchronised algorithm, becomes $\lceil \log_3 \ell \rceil \lambda_2 \bar{t}$. In comparison with the binary search algorithm (whose time-complexity is $\lceil \log_2 \ell \rceil \bar{t}$ since it takes at most $\lceil \log_2 \ell \rceil$ iterations) the synchronised parallel algorithm has been proved inefficient for large λ_p , which usually occurs with large values of p .

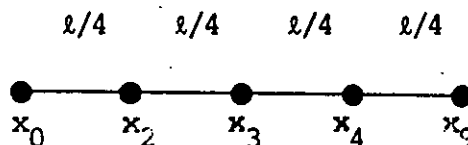
For the analysis of the asynchronous parallel root-finding algorithm we use n to refer to the number of function evaluations completed by the algorithm. Bearing in mind that these computations are performed in parallel, then the expected time complexity is $\bar{n}/2$ as

$n \rightarrow \infty$. Consequently, the speed-up ratio between the sequential binary search and this algorithm is

$$S_2 = \frac{\lceil \log_2 \ell \rceil \bar{t}}{\frac{nt}{2}} = 2 \frac{\lceil \log_2 \ell \rceil}{n}$$

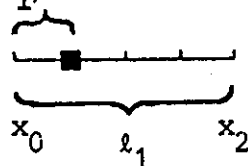
Therefore, the requirement to find the exact value of n is essential. In the worst case, this value is given by the length of the largest path in the transition tree. Analysis of the transition tree carried out by Hayafil and Kung [Hayafil 1975] shows that, in the worst case, the asynchronous algorithm supersedes the synchronised version with two processors when the penalty factor $\lambda_2 > 1.142$.

The asynchronous algorithm introduced can be generalised to three or more processors. For the case of three processors we can start with the following diagram:



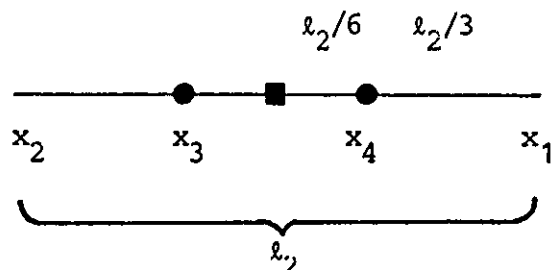
The three processors are activated to evaluate the function at points x_2 , x_3 and x_4 , which are chosen as indicated in the above diagram. As a result of the concurrent function evaluation, without loss of generality, one of the following states will occur:

State 1: $\ell_1/16$



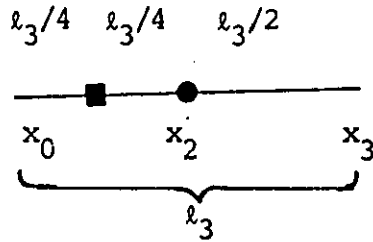
$$\ell_1 = \ell/4$$

State 2:



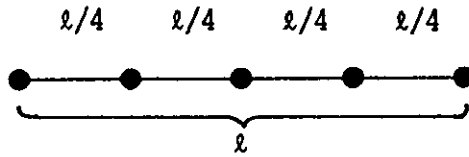
$$\ell_2 = 3\ell/4$$

State 3:

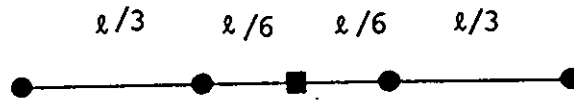


$$l_3 = l/2$$

Where in each case the function is evaluated at point '■'. States 1 and 3 are in fact defining the same pattern.



while State 2 yields the pattern



and an asynchronous algorithm with three processors can be fully defined by using the above two patterns.

In general, $\lfloor p/2 \rfloor + 1$ patterns are sufficient for defining an asynchronous algorithm with p processors.

Another highly important group of methods, the iterative methods are utilised to solve many problems, in particular numerical ones. For example, zeros of a function f can be computed by the Newton iteration:

$$x_{i+1} = x_i + f(x_i)/f'(x_i)$$

also, the solutions of linear systems by iterations of the form:

$$\vec{x}_{i+1} = A \vec{x}_i + \vec{b}$$

where \vec{x}_i , \vec{b} are n -vectors and A is an $(n \times n)$ matrix.

When designing synchronised or asynchronous iterative parallel algorithms, one of two (or a combination of both) strategies can be followed. The first one aims at exploiting the inherent parallelism within the iterative function f , and the second one to exploit the fluctuations in process speed, mentioned earlier in this chapter, utilising more than one processor to compute the same function in parallel.

In synchronised iterative algorithms, iterations are generated just as in a sequential algorithm, except that the iteration function is decomposed so that each iteration step can be executed by more than one process, which are then synchronised at the end of each iteration. Consequently, these algorithms differ from the sequential ones in the execution time required by each iteration. However one must bear in mind that synchronised iterative algorithms are not suitable for those iterative functions which cannot be decomposed into mutually independent tasks of the same complexity.

On the other hand, asynchronous iterative algorithms are free from any form of synchronisation constraints. To design asynchronous iterative parallel algorithms, it is desirable that one should first identify certain variables, such that each step can be regarded as computing the new values of these variables from their old values. In general, the selection of these variables is such that the updating of each of them constitutes a significant portion of the work involved in each iteration. Next one should define concurrent processes that would update these variables asynchronously.

In [Kung 1976] a particular consideration has been given to algorithms purely derived from the second of the previous two

strategies, which are called **simple asynchronous iterative** algorithms. Their main advantage is their general applicability, in other terms they are not restricted to numerical processes only, but they can be used to speed-up any sequence of tasks with a particular attraction when the task decomposition appears to be difficult. There are, however, some disadvantages such as the requirement for critical sections and the fact that the speed-up is quite limited if the fluctuations in computation time are not large.

Finally, Kung [Kung 1976] introduced the special class of the **adaptive asynchronous algorithms** utilising **global dequeues** (i.e. 'double-ended queues' see [Knuth 1969]) to hold the tasks to be executed in parallel. According to this class of algorithms the tasks performed by a particular process are not specified a priori, but depend upon the relative speeds of the processes. The efficiency of such an algorithm is obtained from the fact that processes are able to adjust themselves during computation so that they can all finish in about the same time. Thus, the concept of adaptive algorithms seems to be fundamental to the design of many efficient asynchronous algorithms.

To conclude this paragraph, we assume that synchronised algorithm should be utilised when fluctuations in process speed are small and when there are relatively few processes to be synchronised. On the contrary, asynchronous algorithms are, in general, more efficient than synchronised ones, since processors never waste time in waiting for inputs. Thus, the algorithms can take advantage of running fast processes and they can be adaptive so that the processes can finish at approximately the same time. Furthermore, an asynchronous algorithm can be more reliable than a synchronised one since even if some processes are blocked forever, the algorithm can still continue

computing the solution of the problem as long as no blocking occurs in critical sections and there remains at least one active process.

Chapter 4

PARALLEL SEARCHING ALGORITHMS

4.1 INTRODUCTION

Many computer applications such as database systems, information processing and artificial intelligence have to include a searching function in order to deal with some outstanding problems. Since searching is the most time-consuming part of many of the programs involved, it was necessary to find the best searching method to replace existing poor ones. Consequently, a substantial increase in speed is most likely to be achieved. However, it is sometimes possible to organise the data structure in such a way that the searching can be entirely eliminated. Unfortunately, there are few cases where we do still use the 'poor' searching methods as the only available alternative. For such cases a parallel implementation is much appreciated.

Basically by searching we mean the process of examining the contents of memory locations to see whether they match some given template or keyword. In general, we shall assume that a set of N records has been previously stored on some primary storage devices, and it is required to locate the appropriate one. Algorithms for searching are presented with a so-called argument 'key' and the problem is to locate which record matches that 'key'. After the search algorithm is completed, two possibilities arise:

1. either the search was successful, a record with 'key' as one of its fields' value was located, or
2. the search has failed and the key is nowhere to be found.

Several searching algorithms have been proposed to run on uniprocessor types of computers and a lower bound of $\log N$ has been

established (see [Knuth 1973]). These algorithms could be grouped into two main classes: those dealing with unsorted sets of records, and those dealing with sorted ones. Examples of the first class of algorithms are: the basic sequential search, the self-organising sequential search (either move to front or transpose method). The latter two algorithms are based on rearranging the set of records so that the most frequently accessed keys are quickly located. On the other hand, the second class of algorithms which deals with ordered sets of records, include the binary search, the interpolation search, the interpolation sequential search and the jump search algorithms.

In this chapter we shall design and analyse parallel algorithms for the basic sequential search, the binary search and many jump search algorithms. For the sequential search, two different versions are presented and analysed and for the binary search we proposed three different versions.

4.2 AN MIMD IMPLEMENTATION OF THE SEQUENTIAL SEARCH ALGORITHM

Given a set of unordered records R_1, R_2, \dots, R_n with the respective key values K_1, K_2, \dots, K_n , and given an argument 'key', the sequential search consists of successively accessing a record, R_i , (starting from R_1 and progressing towards R_n , or vice versa) and comparing the argument key with K_i . The search will succeed when there exists i such that $K_i = \text{key}$.

Without prior knowledge about the stored records, they are usually assumed to be uniformly probable (i.e. every record is likely to be the searched one, in other words all the records have the same

probability* of being the sought one), and uniformly accessible. Consequently, on average, the sequential search algorithm would perform:

$$C_N = \frac{1 + 2 + \dots + N}{N}$$

$$= \frac{1}{2} (N+1)$$

Key comparisons for a successful search and

$$C_N = N$$

Key comparisons for an unsuccessful search, since we would be certain that such a key is non existent among the N records only if every possible record is examined [Knuth 1973].

The implementation of the sequential search algorithm on the multiprocessor systems available at Loughborough University is presented below.

The original set of N records is partitioned into M subsets of nearly equal lengths (i.e. N/M elements). Each active processor of the P multiprocessor system would apply the sequential search to locate the key in one of the subsets which it is associated to. For the analysis of the following parallel algorithms we assume that the search finishes after finding the first occurrence of the key and if found it is unique. This means that only one search of the M subsets' searches will be successful. Furthermore, we assume a strict cooperation between the P processors. In the case that the key is located by one of them, a signal is broadcast to all the

* For a set of N records this probability is equal to $1/N$.

remaining processors so that they could immediately stop searching since any further location inspection is obviously redundant. In our MIMD implementation such a signal broadcasting is achieved through a shared boolean variable.

In a no-broadcasting algorithm, once the key was located, all the processors would still carry on searching even if the unique location was already found by one of them. Surely broadcasting would eliminate unnecessary work and thus intuitively would increase the speed-up and efficiency of the search algorithm.

In order to measure the actual performance of the proposed parallel sequential search algorithms, we implemented them on both parallel systems, the Neptune and Balance 8000, and measured their experimental timing. In all three sets of experiments were performed and in each experiment 100 random keys which are uniformly distributed in the set are individually searched and timed. From these timing results we then computed the average speed-ups.

In addition, we also measured the static overheads (i.e. the shared data access overhead - 'SDO' and the parallel control overhead - 'PCO' which are obtained by running two sequential load programs compiled using the commands XPFCLS and XPFCLN* respectively (see Chapter 3).

If T_S and T_N measure the execution of the programs produced respectively by XPFCLS and XPFCLN, then the measured static overheads are computed as indicated below:

* XPFCLS produces a sequential load program where the shared data will be loaded in the shared memory. XPFCLN produces a similar sequential program as XPFCLS except that the shared data will be considered as local and loaded in the local memory

$$\text{SDO \%} = \frac{T_S - T_N}{T_1} * 100$$

and

$$\text{PCO \%} = \frac{T_1 - T_S}{T_1} * 100.$$

One of the first decisions we are faced with is the selection of the number of subsets M which could vary from P to N so that the processing time is optimal. In the following paragraph we shall attempt to show that as far as the parallel searching algorithm is concerned, it is always best to choose M equal to P . For this purpose, we shall first compute the average time-complexity of the parallel sequential search algorithm when $M=P$ and secondly solve the following inequality:

$$\exists M^*, M^* > P / T_p(M^*) < T_p(P)$$

where $T_p(M^*)$ is the time-complexity of the parallel sequential algorithm when $M=M^*$ with M^* unknown.

We also use two additional costs C_1 and C_2 which represent the cost of a single key comparison and the cost of acquiring a subset (i.e. accessing a parallel path).

When the set is divided into P subsets, each containing $\frac{N}{P}$ elements, the average time-complexity T_p is computed as the product of the average number of key comparisons by C_1 plus PC_2 . Since each processor searches a subset, then on average, each processor would perform:

$$\frac{1}{N/P} \sum_{i=1}^{N/P} i = \frac{N+P}{2P} \text{ key comparisons}$$

Consequently

$$T_p (M=P) = \frac{N+P}{2P} C_1 + PC_2$$

On the other hand, if we select $M > P$, we end up with each processor processing at most $\frac{M}{P}$ subsets, each containing $\frac{N}{M}$ elements. Now searching a subset of $\frac{N}{M}$ elements requires, on average, $\frac{N+M}{2M}$ key comparisons. If a processor finds the key on the j th subset it is allocated, then it would have performed

$$(j-1) \frac{N}{M} + \frac{N+M}{2M} \text{ key comparisons}$$

The average of the above expression over all the $\frac{M}{P}$ subsets processed by a processor is:

$$\frac{N+P}{2P}$$

Note that it is exactly the same result as that of the above case ($M=P$). However since M paths are executed, the path scheduling ring structure would be accessed M times, therefore the average time-complexity is

$$T_p (M > P) = \frac{N+P}{2P} C_1 + M C_2$$

Since $M > P$, it implies that

$$T_p (M > P) > T_p (M=P)$$

This concludes our proof that as far as the parallel sequential search is concerned, it is best to divide the set into P .

Tables 4.1, 4.2 and 4.3 report the experimental timing of the parallel sequential search algorithm measured when searching for keys located at 1, middle of a subset and N . For all these experiments, M , the number of subsets was varied from 4, 8, ..., 512. As can be seen from the tables, the value of T_p increases when M increases except for $M=16$ and when three processors are in use. Note that this particular case is a special one and is not to be considered as a counter-example. Actually the T_3 values did not decrease when M increased to 16 but it did increase by a factor higher than expected for $M=8$ due to the fact that the difference in the number of paths executed by each processor is greater than the other cases. More specifically for the case of 8 paths ($M=8$), two processors did execute two paths each but the third executed four paths. By comparison with the case of $M=16$ and three processors, this difference is only equal to one path.

From the experimental results as shown in Tables 4.1, 4.2 and 4.3 we noticed that as M increases, the speed-up (relative to the case of $M=4$) decreases, a fact that was proven previously. Thus our experiments confirmed that no major gains in efficiency can be achieved if the set is partitioned into more than P subsets.

In the following sections we shall design and analyse parallel sequential search with and without broadcasting when the number of subsets is equal to P . For the former case two different versions of the parallel search are considered (referred to as versions 1.0 and 2.0). All algorithms partition the original set into P subsets, but from the point of view of the contents of the subsets, the two groups of subsets are quite different.

M	T1	T2	T3	T4	TS	TN	SP2	SP3	SP4	SDO	PCO
4	4.15	2.77	1.42	1.43	4.14	4.13	1.50	2.92	2.90	0.24	0.24
8	4.84	2.78	2.80	1.42	4.82	4.81	1.49	1.48	2.92	0.24	0.48
16	5.18	2.77	1.77	1.42	5.16	5.15	1.50	2.34	2.92	0.24	0.48
32	5.39	2.79	1.91	1.42	5.36	5.33	1.49	2.17	2.92	0.72	0.72
64	5.50	2.80	1.88	1.43	5.45	5.42	1.48	2.21	2.90	0.72	1.20
128	5.61	2.83	1.92	1.45	5.50	5.46	1.47	2.16	2.86	0.96	2.65
256	5.77	2.89	1.95	1.47	5.55	5.50	1.44	2.13	2.82	1.20	5.30
512	6.03	3.03	2.03	1.53	5.57	5.54	1.37	2.04	2.71	0.72	11.08

TABLE 4.1: EXPERIMENTAL PERFORMANCE MEASUREMENTS OF THE PARALLEL SEARCH ALGORITHM WHEN SEARCHING FOR K(1).

M	T1	T2	T3	T4	TS	TN	SP2	SP3	SP4	SDO	PCO
4	4.34	2.53	1.88	1.29	4.33	4.32	1.72	2.31	3.36	0.23	0.23
8	4.69	2.56	1.93	1.31	4.68	4.64	1.70	2.25	3.31	0.92	0.23
16	4.82	2.56	1.72	1.28	4.81	4.77	1.70	2.52	3.39	0.92	0.23
32	4.90	2.55	1.75	1.30	4.88	4.84	1.70	2.48	3.34	0.92	0.46
64	5.00	2.57	1.71	1.30	4.95	4.91	1.69	2.54	3.34	0.92	1.15
128	5.09	2.59	1.75	1.32	4.98	4.95	1.68	2.48	3.29	0.69	2.53
256	5.21	2.64	1.78	1.35	4.98	4.95	1.64	2.44	3.21	0.69	5.30
512	5.47	2.75	1.85	1.40	5.02	4.99	1.58	2.35	3.10	0.69	10.37

TABLE 4.2: EXPERIMENTAL PERFORMANCE MEASUREMENTS OF THE PARALLEL SEARCH ALGORITHM WHEN SEARCHING FOR THE KEY LOCATED AT THE MIDDLE POSITION OF THE SET.

M	T1	T2	T3	T4	TS	TN	SP2	SP3	SP4	SDO	PCO
4	4.98	2.55	2.55	1.28	4.93	4.90	1.95	1.95	3.89	0.60	1.00
8	4.99	2.54	1.91	1.29	4.98	4.93	1.96	2.61	3.86	1.00	0.20
16	4.98	2.54	1.88	1.30	4.96	4.93	1.96	2.65	3.83	0.60	0.40
32	5.01	2.56	1.76	1.29	4.98	4.93	1.95	2.83	3.86	1.00	0.60
64	5.04	2.57	1.75	1.30	4.98	4.94	1.94	2.85	3.83	0.80	1.20
128	5.10	2.59	1.75	1.30	4.99	4.94	1.92	2.85	3.83	1.00	2.21
256	5.22	2.64	1.77	1.34	4.99	4.96	1.89	2.81	3.72	0.60	4.62
512	5.45	2.75	1.85	1.39	5.02	4.97	1.81	2.69	3.58	1.00	8.63

TABLE 4.3: EXPERIMENTAL PERFORMANCE MEASUREMENTS OF THE PARALLEL SEARCH ALGORITHM WHEN SEARCHING FOR $K(N)$.

In mathematical terms, if processors are numbered by k , $k = 1, 2, \dots, P$ then processor k would access the following locations if

$$i) \text{ version 1.0} \quad (k-1) \frac{N}{P} + i$$

$$ii) \text{ version 2.0} \quad (i-1) P + k$$

4.2.1 PARALLEL SEQUENTIAL SEARCH WITHOUT BROADCASTING

In this algorithm the set of records is partitioned into P subsets, each of which contains $\frac{N}{P}$ elements. Each processor searches a subset until either it finds the key or the subset is fully exhausted. Since the key is unique, only one of the P activated processors would find it whereas all the others would perform exactly $\frac{N}{P}$ key comparisons. Therefore, when P processors are being used, the time-complexity of the algorithm is

$$T_p = \frac{N}{P}$$

On the other hand, if only one processor is used, the time complexity T_1 is given by

$$T_1 = j \quad \text{where } j = 1, 2, \dots, N$$

which is also equal to the following expression, if we consider the P subsets individually

$$T_1 = (k-1) \frac{N}{P} + i$$

where $k = 1, 2, \dots, P$

$$i = 1, 2, \dots, \frac{N}{P}$$

Since $i \leq \frac{N}{P}$, we have

$$T_1 \leq (k-1) \frac{N}{P} + \frac{N}{P}$$

$$\leq k \frac{N}{P}$$

The speed-up $S_p(k)$ is then

$$S_p(k) = \frac{T_1(k)}{T_p} \leq \frac{k \frac{N}{P}}{\frac{N}{P}}$$

or

$$S_p(k) \leq k, \quad k = 1, 2, \dots, P$$

Averaging the above expression over the P subsets we get the average speed-up S_p of the parallel search algorithm without broadcasting as:

$$S_p \leq \frac{1}{P} \sum_{k=1}^P k$$

$$\leq \frac{P+1}{2}$$

and the efficiency E_p as

$$E_p = \frac{S_p}{P}$$

$$\frac{P+1}{2P}$$

In Table 4.4 we present the experimental timing results of the parallel sequential search algorithm with no broadcasting. These results, as it is seen, are in close agreement with the predicted ones.

TABLE 4.4: EXPERIMENTAL TIMING RESULTS OF THE PARALLEL SEQUENTIAL SEARCH ALGORITHM WITHOUT BROADCASTING PERFORMED ON THE BALANCE 8000 SYSTEM

T1	T2	T3	T4	T5	SP2	SP3	SP4	SP5	T1	T2	T3	T4	T5	SP2	SP3	SP4	SP5
.01	.65	.46	.33	.29	.02	.02	.03	.03	.55	.54	.41	.29	.23	1.02	1.34	1.90	2.39
.01	.56	.38	.28	.24	.02	.03	.04	.04	.52	.55	.37	.28	.24	.95	1.41	1.86	2.17
.02	.55	.38	.28	.24	.04	.05	.07	.08	.55	.55	.38	.29	.23	1.00	1.45	1.90	2.39
.04	.55	.38	.29	.24	.07	.11	.14	.17	.55	.54	.37	.29	.24	1.02	1.49	1.90	2.29
.07	.56	.38	.28	.24	.12	.18	.25	.29	.56	.54	.38	.29	.23	1.04	1.47	1.93	2.43
.07	.54	.37	.30	.23	.13	.19	.23	.30	.59	.54	.37	.29	.23	1.09	1.59	2.03	2.57
.08	.55	.37	.29	.23	.15	.22	.28	.35	.59	.54	.50	.28	.23	1.09	1.18	2.11	2.57
.08	.55	.38	.29	.24	.15	.21	.28	.33	.59	.54	.38	.29	.21	1.09	1.55	2.03	2.81
.09	.51	.38	.29	.23	.18	.24	.31	.39	.61	.54	.37	.29	.23	1.13	1.65	2.10	2.65
.09	.55	.38	.30	.23	.16	.24	.30	.39	.62	.55	.36	.29	.23	1.13	1.72	2.14	2.70
.11	.56	.38	.29	.24	.20	.29	.38	.46	.64	.54	.35	.29	.25	1.19	1.83	2.21	2.56
.10	.55	.34	.29	.24	.18	.29	.34	.42	.63	.55	.37	.29	.23	1.15	1.70	2.17	2.74
.11	.55	.38	.29	.24	.20	.29	.38	.46	.67	.54	.49	.28	.25	1.24	1.37	2.39	2.68
.11	.54	.38	.30	.23	.20	.29	.37	.48	.67	.55	.39	.29	.23	1.22	1.72	2.31	2.91
.13	.51	.37	.25	.24	.25	.35	.52	.54	.68	.54	.37	.29	.24	1.26	1.84	2.34	2.83
.14	.56	.38	.29	.23	.25	.37	.48	.61	.72	.54	.38	.29	.23	1.33	1.89	2.48	3.13
.15	.55	.38	.28	.24	.27	.39	.54	.63	.71	.55	.38	.28	.24	1.29	1.87	2.54	2.96
.17	.55	.38	.29	.18	.31	.45	.59	.94	.71	.53	.37	.29	.23	1.34	1.92	2.45	3.09
.16	.55	.37	.29	.24	.29	.43	.55	.67	.72	.54	.38	.31	.24	1.33	1.89	2.32	3.00
.21	.50	.38	.29	.21	.42	.55	.72	1.00	.73	.54	.49	.40	.23	1.35	1.49	1.83	3.17
.23	.56	.31	.30	.23	.41	.74	.77	1.00	.74	.54	.37	.30	.24	1.37	2.00	2.47	3.08
.23	.56	.39	.29	.24	.41	.59	.79	.96	.75	.54	.37	.28	.22	1.39	2.03	2.68	3.41
.24	.55	.38	.28	.23	.44	.63	.86	1.04	.76	.53	.38	.28	.22	1.43	2.00	2.71	3.45
.27	.55	.38	.29	.24	.49	.71	.93	1.13	.76	.54	.38	.28	.23	1.41	2.00	2.71	3.30
.27	.46	.37	.29	.23	.59	.73	.93	1.17	.77	.55	.37	.30	.23	1.40	2.08	2.57	3.35
.31	.55	.38	.25	.24	.56	.82	1.24	1.29	.77	.54	.39	.38	.23	1.43	1.97	2.03	3.35
.31	.55	.38	.29	.24	.56	.82	1.07	1.29	.79	.54	.49	.38	.23	1.46	1.61	2.08	3.43
.33	.56	.34	.28	.23	.59	.97	1.18	1.43	.79	.54	.38	.30	.23	1.46	2.08	2.63	3.43
.34	.55	.38	.29	.24	.62	.89	1.17	1.42	.80	.55	.37	.40	.23	1.45	2.16	2.00	3.48
.37	.62	.37	.29	.23	.60	1.00	1.28	1.61	.81	.52	.34	.40	.25	1.56	2.38	2.02	3.24
.37	.55	.38	.29	.24	.67	.97	1.28	1.54	.82	.54	.38	.29	.23	1.52	2.16	2.83	3.57
.38	.55	.37	.28	.23	.69	1.03	1.36	1.65	.83	.55	.37	.29	.24	1.51	2.24	2.86	3.46
.38	.54	.38	.29	.25	.70	1.00	1.31	1.52	.83	.55	.37	.28	.23	1.51	2.24	2.96	3.61
.39	.55	.37	.29	.23	.71	1.05	1.34	1.70	.87	.54	.50	.28	.24	1.61	1.74	3.11	3.63
.40	.60	.37	.29	.24	.67	1.08	1.38	1.67	.87	.54	.39	.29	.23	1.61	2.23	3.00	3.78
.42	.55	.38	.28	.23	.76	1.11	1.50	1.83	.87	.54	.37	.29	.24	1.61	2.35	3.00	3.63
.41	.54	.37	.28	.23	.76	1.11	1.46	1.78	.90	.54	.38	.29	.24	1.67	2.37	3.10	3.75
.41	.55	.38	.28	.24	.75	1.08	1.46	1.71	.93	.54	.37	.37	.23	1.72	2.51	2.51	4.04
.43	.55	.38	.29	.23	.78	1.13	1.48	1.87	.93	.54	.38	.29	.23	1.72	2.45	3.21	4.04
.44	.57	.37	.28	.23	.77	1.19	1.57	1.91	.93	.54	.38	.40	.23	1.72	2.45	2.33	4.04
.45	.55	.38	.29	.23	.82	1.18	1.55	1.96	.99	.55	.49	.41	.25	1.80	2.02	2.41	3.96
.45	.55	.37	.29	.24	.82	1.22	1.55	1.88	.99	.55	.39	.33	.33	1.80	2.54	3.00	3.00
.47	.55	.37	.29	.23	.85	1.27	1.62	2.04	1.00	.61	.37	.28	.33	1.64	2.70	3.57	3.03
.47	.56	.38	.28	.24	.84	1.24	1.68	1.96	1.03	.54	.38	.30	.29	1.91	2.71	3.43	3.55
.47	.56	.38	.29	.23	.84	1.24	1.62	2.04	1.03	.54	.38	.28	.24	1.91	2.71	3.68	4.29
.48	.55	.37	.29	.24	.87	1.30	1.66	2.00	1.06	.54	.37	.38	.23	1.96	2.86	2.79	4.61
.50	.54	.39	.29	.23	.93	1.28	1.72	2.17	1.06	.54	.39	.33	.24	1.96	2.72	3.21	4.42
.51	.55	.37	.28	.24	.93	1.38	1.82	2.13	1.06	.55	.35	.29	.23	1.93	3.03	3.66	4.61
.50	.55	.38	.29	.23	.91	1.32	1.72	2.17	AVERAGE								
.51	.58	.38	.27	.24	.88	1.34	1.89	2.13									
.52	.55	.38	.29	.23	.95	1.37	1.79	2.26									
.53	.55	.51	.29	.24	.96	1.04	1.83	2.21	.52	.55	.38	.30	.24	.95	1.36	1.73	2.20

4.2.2 PARALLEL SEQUENTIAL SEARCH WITH BROADCASTING

Cooperation among concurrent processes is one of the major key features to achieving efficient parallel algorithms. In particular, for our searching problem where the key is supposed to be unique, processors must maintain a certain degree of cooperation between themselves so that the current state of the search is known by every processor at every time. Such information when used appropriately, leads to an efficient searching algorithm since once the key is located by one processor, all the remaining ones also stop searching and thus avoid unnecessary key comparisons.

In the following, we shall consider two different versions which we call versions 1.0 and 2.0 for the parallel sequential search algorithm with broadcasting. Both versions partition the original set among the P available processors and then allocate each processor to search one subset using the traditional sequential search algorithm. However the two versions differ from each other in that the elements accessed by one processor in one version are not the same elements accessed by the same processor in the other version. In other words, if processors are numbered by k , where $k = 1, 2, \dots, P$ then processor k would access the following locations:

$$(k-1) \frac{N}{P} + i$$

in the case of version 1.0 and

$$(i-1) P + k, \text{ otherwise}$$

where $i, i = 1, 2, \dots, \frac{N}{P}$ represents the iteration number. Tables 4.5 and 4.6 list explicitly all the locations accessed by each processor k for each iteration i respectively for version 1.0 and version 2.0.

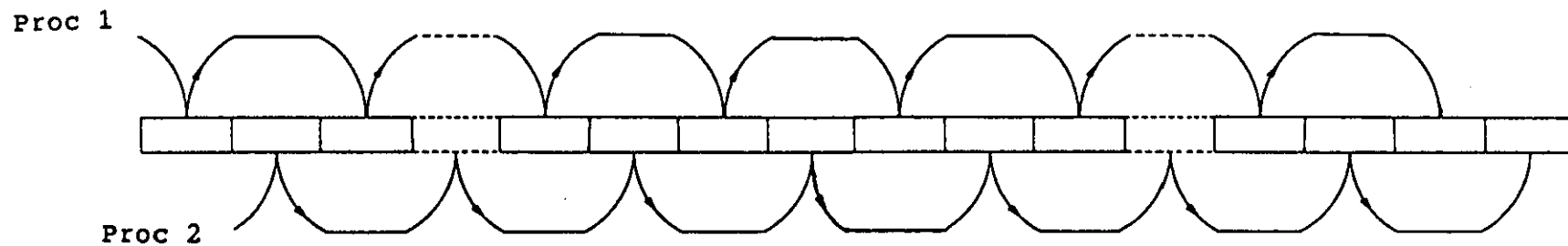
From the algorithmic point of view, when the algorithm is executed in parallel, P key comparisons are performed during every iteration i , however the inspected locations in version 1.0 differ from those inspected in version 2.0. More specifically, the P locations are consecutive in version 2.0 whereas they are distant by P in version 1.0. Figure 4.1 illustrates these points when $N=32$ and $P=2$

1) Analysis of Version 1.0

The set of unordered records is partitioned into P subsets according to the indexing relative to version 1.0. Each subset contains $\frac{N}{P}$ items (for simplicity purposes, we select N a multiple of P). It is a common practice that during the computational analysis, all the static and dynamic overheads are deliberately ignored since it is a difficult problem to include them. For these reasons we decided to ignore these overheads.

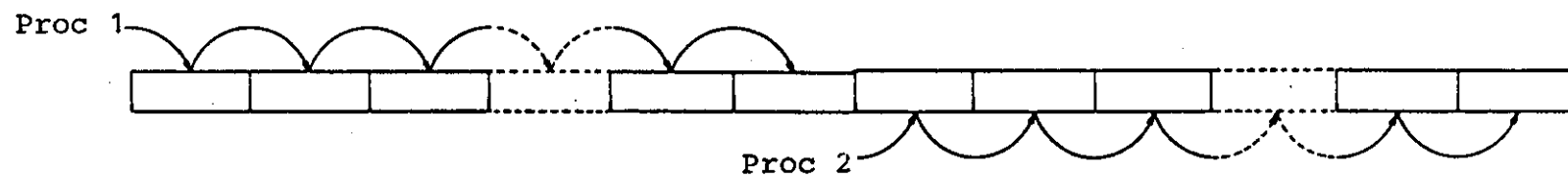
Using a single processor to execute the parallel algorithm version 1.0, the time-complexity T_1 is proportional to the number of keys inspected. If $j, j = 1, 2, \dots, N$ is the location where the target key is to be found then:

$$T_1 = j \quad \text{where } j = 1, 2, \dots, N$$



(a) Version 2.0

FIGURE 4.1: PARALLEL SEQUENTIAL SEARCH ALGORITHM WITH BROADCASTING.



(b) Version 1.0

Proc.	Locations accessed			
1	1	2	...	$\frac{N}{P}$
2	$\frac{N}{P} + 1$	$\frac{N}{P} + 2$...	$2\frac{N}{P}$
⋮	⋮	⋮		⋮
k	$(k-1)\frac{N}{P} + 1$	$(k-1)\frac{N}{P} + 2$...	$k\frac{N}{P}$
⋮	⋮	⋮		⋮
P	$(P-1)\frac{N}{P} + 1$	$(P-1)\frac{N}{P} + 2$...	$P\frac{N}{P}$

TABLE 4.5: LOCATION INDEXES ACCESSED BY THE PARALLEL SEQUENTIAL SEARCH ALGORITHM VERSION 1.0 PER EVERY PROCESSOR

Proc.	Locations accessed			
1	1	P+1	...	$(\frac{N}{P} - 1)P + 1$
2	2	P+2	...	$(\frac{N}{P} - 1)P + 2$
⋮	⋮	⋮		⋮
k	k	P+k	...	$(\frac{N}{P} - 1)P + k$
⋮	⋮	⋮		⋮
P	P	P+P	...	$(\frac{N}{P} - 1)P + P$

TABLE 4.6: LOCATION INDEXES ACCESSED BY THE PARALLEL SEQUENTIAL SEA ALGORITHM VERSION 2.0 PER EVERY PROCESSOR

Since we assumed the existence (only successful searches are considered*), and uniqueness (only single-key search for multiple-key search analysis is quite similar to the unsuccessful case) of the key argument, then only one processor will locate the key after a certain number of key comparisons. Note that the number of iterations is also the number of keys compared by each processor since P such keys are compared every iteration

$$T_p = i \quad (4.1)$$

and

$$T_1 = (k-1) \frac{N}{P} + i$$

where $i = 1, 2, \dots, \frac{N}{P}$
and $k = 1, 2, \dots, P$.

From both equations defined in (4.1) we compute the speed-up $S_p(i,k)$ for every key found during iteration i by processor k as

$$S_p(i,k) = \frac{T_1}{T_p} = 1 + \frac{N}{P} (k-1) \frac{1}{i} \quad (4.2)$$

The average of the above expression over all the values $i=1, 2, \dots, \frac{N}{P}$ yields the average speed-up over all keys found by processor k . We denote such an average speed-up by $S_p(k)$ where

$$S_p(k) = \frac{1}{N/P} \sum_{i=1}^{N/P} S_p(i,k)$$

* The analysis of the case when the search is unsuccessful is a straightforward operation.

Substituting $S_p(i,k)$ by its value as defined in (4.2) and computing the corresponding summation, we obtain the following

$$S_p(k) = 1 + (k-1) \sum_{i=1}^{N/P} \frac{1}{i} \quad (4.3)$$

An upper bound for $\sum 1/i$ is derived as follows:

$$\sum_{i=1}^{N/P} \frac{1}{i} < 1 + \int_1^{N/P} \frac{dx}{x}$$

then

$$\begin{aligned} \sum_{i=1}^{N/P} \frac{1}{i} &< 1 + \ln \frac{N}{P} - \ln 1 \\ &< 1 + \ln \frac{N}{P} \end{aligned} \quad (4.4)$$

where \ln is log base e .

Reporting (4.4) in (4.3) we get

$$S_p(k) < 1 + (1 + \ln \frac{N}{P})(k-1) \quad (4.5)$$

Finally the average speed-up of this algorithm is obtained if we average $S_p(k)$ over all the values for k

$$S_p = \frac{1}{P} \sum_{k=1}^P S_p(k)$$

Substituting $S_p(k)$ by its equivalent value defined in (4.5) we get an upper bound for the parallel sequential search algorithm average speed-up as

$$S_p < \frac{P-1}{2} * (1 + \ln \frac{N}{P}) + 1$$

In Figure 4.2, we plotted the theoretical speed-up of version 1.0 versus the number of processors which is **superlinear*** for $P < \frac{N}{e}$ and linear otherwise. Such a superlinear speed-up is logically achievable since broadcasting 'kills' off unnecessary searches. For instance, consider the problem of searching a set of N records, $N = 8192$ for a key which is located at 4097. Sequentially, the time complexity of the algorithm is proportional to

$$T_1 = 4097$$

Now using two processors, this key would be found by one of the processors after one single key comparison. According to our broadcasting assumption, the second processor would also make one key comparison.

Therefore, the time complexity T_2 is:

$$T_2 = 1$$

Consequently, the speed-up for this particular example is

$$S_p = \frac{T_1}{T_2} = \frac{4097}{1} = 4097$$

which is greater than 2.

In Table 4.7 we present the experimental timing results of the parallel sequential search algorithm version 1.0. As predicted, the average speed-up is superlinear but not as high as those theoretical

* When the speed-up exceeds the number of processors used (see [Quinn 1987] for a discussion and examples of algorithms that achieve a superlinear speed-up.

FIGURE 4.2: THEORETICAL SPEED-UP OF THE PARALLEL SEQUENTIAL SEARCH ALGORITHM VERSION 1.0

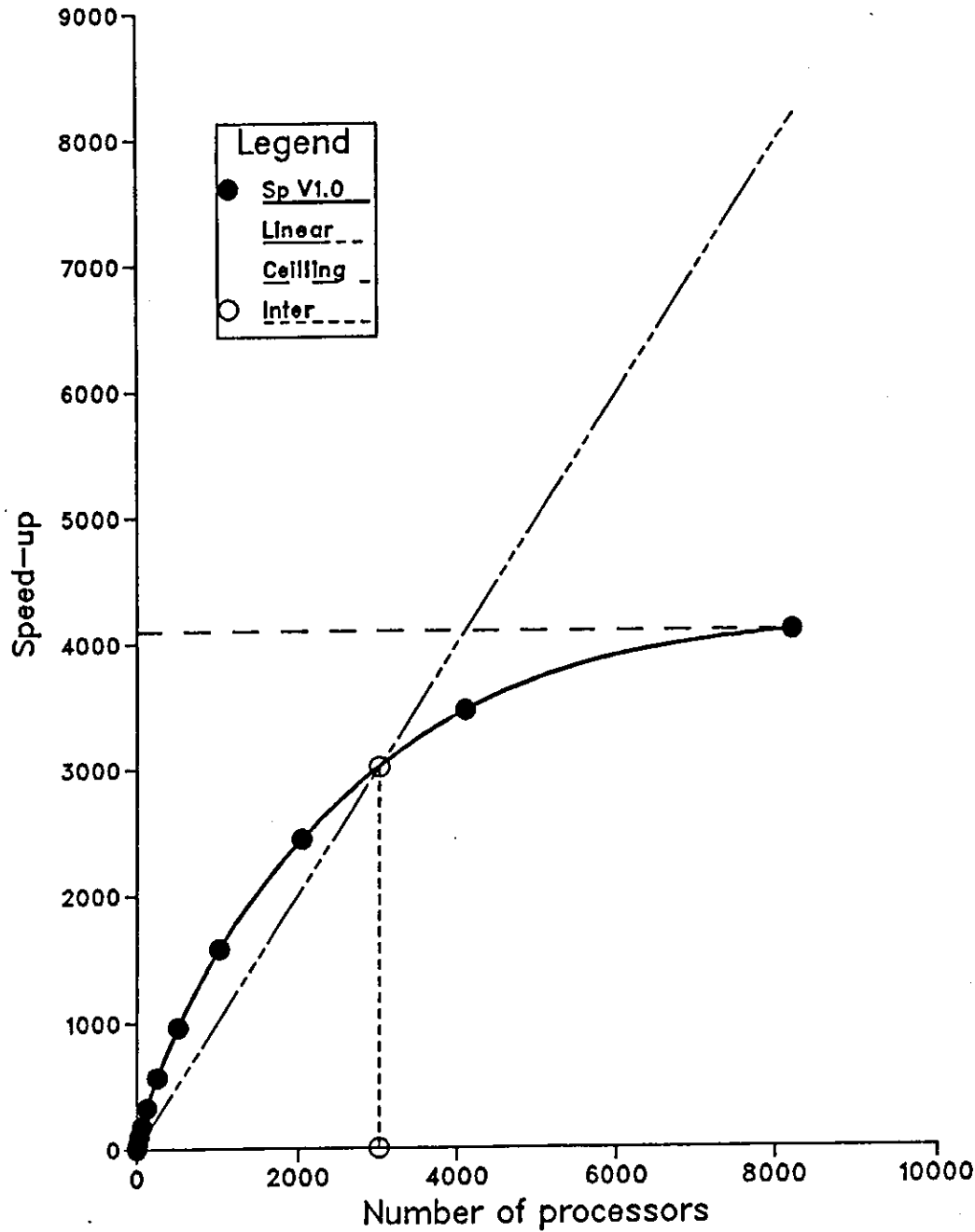


TABLE 4.7: THE EXPERIMENTAL TIMING RESULTS OF THE PARALLEL SEQUENTIAL SEARCH ALGORITHM VERSION 1.0

T1	T2	T3	T4	T5	SP2	SP3	SP4	SP5	T1	T2	T3	T4	T5	SP2	SP3	SP4	SP5
.01	.03	.04	.04	.05	.33	.25	.25	.20	.70	.02	.25	.01	.16	35.00	2.80	70.00	4.38
.02	.03	.05	.04	.05	.67	.40	.50	.40	.71	.02	.26	.03	.17	35.50	2.73	23.67	4.18
.03	.05	.05	.05	.05	.60	.60	.60	.60	.72	.04	.27	.03	.18	18.00	2.67	24.00	4.00
.05	.07	.06	.07	.07	.71	.83	.71	.71	.73	.04	.27	.05	.19	18.25	2.70	14.60	3.84
.09	.11	.11	.10	.12	.82	.82	.90	.75	.74	.04	.30	.06	.20	18.50	2.47	12.33	3.70
.09	.11	.12	.11	.11	.82	.75	.82	.82	.77	.09	.32	.10	.22	8.56	2.41	7.70	3.50
.10	.13	.12	.13	.12	.77	.83	.77	.83	.78	.08	.32	.09	.20	9.75	2.44	8.67	3.90
.11	.12	.13	.12	.13	.92	.85	.92	.85	.78	.09	.32	.10	.23	8.67	2.44	7.80	3.39
.11	.14	.13	.14	.13	.79	.85	.79	.85	.80	.11	.31	.13	.25	7.27	2.58	6.15	3.20
.12	.14	.14	.14	.13	.86	.86	.86	.92	.83	.14	.37	.14	.26	5.93	2.24	5.93	3.19
.14	.15	.16	.15	.16	.93	.88	.93	.88	.83	.14	.38	.14	.29	5.93	2.18	5.93	2.86
.14	.17	.16	.16	.17	.82	.88	.88	.82	.84	.15	.40	.16	.30	5.60	2.10	5.25	2.80
.14	.16	.16	.16	.16	.88	.88	.88	.88	.89	.19	.43	.20	.07	4.68	2.07	4.45	12.71
.15	.17	.17	.17	.17	.88	.88	.88	.88	.89	.21	.45	.20	.06	4.24	1.98	4.45	14.83
.17	.19	.19	.19	.19	.89	.89	.89	.89	.89	.22	.46	.22	.09	4.05	1.93	4.05	9.89
.17	.20	.20	.20	.20	.85	.85	.85	.85	.91	.22	.39	.24	.09	4.14	2.33	3.79	10.11
.21	.23	.23	.23	.23	.91	.91	.91	.91	.95	.25	.02	.25	.11	3.80	47.50	3.80	8.64
.22	.23	.23	.23	.23	.96	.96	.96	.96	.95	.26	.03	.27	.12	3.65	31.67	3.52	7.92
.22	.24	.24	.25	.24	.92	.92	.88	.92	.96	.26	.04	.27	.12	3.69	24.00	3.56	8.00
.28	.30	.30	.29	.30	.93	.93	.97	.93	.97	.29	.04	.27	.14	3.34	24.25	3.59	6.93
.29	.32	.32	.32	.04	.91	.91	.91	7.25	.98	.28	.07	.26	.16	3.50	14.00	3.77	6.13
.30	.32	.32	.32	.04	.94	.94	.94	7.50	1.00	.31	.07	.31	.18	3.23	14.29	3.23	5.56
.32	.34	.33	.33	.05	.94	.97	.97	6.40	1.01	.31	.09	.33	.14	3.26	11.22	3.06	7.21
.36	.37	.38	.34	.09	.97	.95	1.06	4.00	1.02	.32	.09	.33	.19	3.19	11.33	3.09	5.37
.36	.39	.38	.04	.10	.92	.95	9.00	3.60	1.02	.33	.10	.34	.19	3.09	10.20	3.00	5.37
.41	.43	.44	.09	.15	.95	.93	4.56	2.73	1.03	.33	.11	.34	.20	3.12	9.36	3.03	5.15
.42	.44	.44	.09	.16	.95	.95	4.67	2.63	1.02	.34	.11	.33	.21	3.00	9.27	3.09	4.86
.44	.46	.46	.14	.17	.96	.96	3.14	2.59	1.05	.36	.12	.37	.22	2.92	8.75	2.84	4.77
.45	.46	.46	.10	.17	.98	.98	4.50	2.65	1.07	.38	.13	.03	.24	2.82	7.13	35.67	4.46
.49	.50	.04	.16	.20	.98	12.25	3.06	2.45	1.07	.37	.15	.03	.24	2.89	7.13	35.67	4.46
.49	.50	.04	.15	.22	.98	12.25	3.27	2.23	1.12	.40	.18	.05	.28	2.80	6.22	22.40	4.00
.50	.52	.04	.16	.24	.96	12.50	3.13	2.08	1.10	.41	.19	.05	.28	2.68	5.79	22.00	3.93
.50	.52	.07	.18	.25	.96	7.14	2.78	2.00	1.10	.42	.19	.06	.29	2.62	5.79	18.33	3.79
.51	.53	.06	.18	.26	.96	8.50	2.83	1.96	1.14	.45	.23	.09	.03	2.53	4.96	12.67	38.00
.54	.56	.08	.20	.28	.96	6.75	2.70	1.93	1.15	.57	.24	.11	.05	2.02	4.79	10.45	23.00
.55	.57	.08	.21	.28	.96	6.88	2.62	1.96	1.16	.39	.24	.11	.04	2.97	4.83	10.55	29.00
.54	.57	.09	.22	.29	.95	6.00	2.45	1.86	1.18	.49	.25	.13	.07	2.41	4.72	9.08	16.86
.54	.57	.11	.22	.30	.95	4.91	2.45	1.80	1.23	.53	.30	.17	.11	2.32	4.10	7.24	11.18
.57	.59	.12	.24	.29	.97	4.75	2.38	1.97	1.23	.53	.26	.18	.11	2.32	4.73	6.83	11.18
.58	.59	.14	.25	.03	.98	4.14	2.32	19.33	1.24	.54	.30	.20	.12	2.30	4.13	6.20	10.33
.60	.61	.14	.25	.07	.98	4.29	2.40	8.57	1.33	.62	.40	.27	.19	2.15	3.33	4.93	7.00
.59	.61	.13	.25	.05	.97	4.54	2.36	11.80	1.32	.63	.50	.29	.22	2.10	2.64	4.55	6.00
.60	.62	.14	.20	.06	.97	4.29	3.00	10.00	1.34	.64	.41	.30	.21	2.09	3.27	4.47	6.38
.62	.63	.17	.28	.07	.98	3.65	2.21	8.86	1.35	.66	.36	.32	.25	2.05	3.75	4.22	5.40
.62	.63	.16	.29	.08	.98	3.88	2.14	7.75	1.34	.71	.45	.32	.26	1.89	2.98	4.19	5.15
.64	.66	.21	.30	.08	.97	3.05	2.13	8.00	1.40	.70	.44	.31	.29	2.00	3.18	4.52	4.83
.66	.68	.20	.31	.10	.97	3.30	2.13	6.60	1.40	.70	.49	.35	.29	2.00	2.86	4.00	4.83
.67	.69	.22	.33	.13	.97	3.05	2.03	5.15	1.41	.71	.61	.37	.29	1.99	2.31	3.81	4.86
.70	.67	.22	.35	.13	1.04	3.18	2.00	5.38	AVERAGE								
.67	.69	.22	.34	.13	.97	3.05	1.97	5.15									
.69	.71	.18	.35	.14	.97	3.83	1.97	4.93									
.70	.71	.24	.31	.16	.99	2.92	2.26	4.38	.69	.37	.22	.20	.17	3.28	4.96	5.83	5.52

values since the overheads were not included in the mathematical model.

The measured static overheads (see Table 4.8) as obtained from the Neptune system are very negligible. Now due to the fluctuation in the timings, we have obtained 'negative' (i.e. less than zero) overhead figures. To obtain accurate overhead measurements we need to measure T_1 , T_S and T_N in exactly the same experimental environment. However, due to the small timing nature of our algorithms (round about a few seconds), these fluctuations are unavoidable and consequently, an overhead with a minus sign should not be interpreted as a gain but as a loss which could not be measured accurately.

ii) Analysis of Version 2.0

As with the previous method, it is obvious to see that every record is accessed by only one processor at a time. If k is the number of a processor, $k = 1, 2, \dots, P$, then each processor will access for every iteration i , $i = 1, 2, \dots, \frac{N}{P}$, the following location

$$(i-1)P + k \quad (4.6)$$

where $i = 1, 2, \dots, \frac{N}{P}$
and $k = 1, 2, \dots, P$.

Thus, using a single processor, the time-complexity is also equal to the number of key comparisons which in this case is equal to the expression defined in (4.6.). The time-complexity of the algorithm when P processors are being used is still equal to the number of iterations performed and defined previously in (4.1). Consequently, the speed-up for every key found by processor k is

TABLE 4.8: THE MEASURED STATIC OVERHEADS OF THE PARALLEL SEQUENTIAL ALGORITHM VERSION 1.0

IKEY	T1	TS	TN	SDO	PCO	IKEY	T1	TS	TN	SDO	PCO
125	.900	.900	.900	0.00	0.00	4302	30.300	29.900	30.200	.99	.33
224	1.600	1.500	1.500	0.00	6.25	4405	30.900	30.500	30.700	.60	.65
300	2.100	2.100	2.100	0.00	0.00	4265	31.800	31.400	31.600	.63	.63
469	2.200	3.100	3.100	0.00	3.13	4590	31.900	31.600	31.800	.63	.31
483	3.300	3.200	3.200	0.00	3.03	4791	32.500	32.100	32.400	.92	.31
661	4.400	4.400	4.400	0.00	0.00	4712	32.600	32.300	32.500	.61	.31
681	4.600	4.500	4.600	2.17	0.00	4825	33.700	33.000	33.200	.60	.30
730	4.900	4.800	4.800	0.00	2.04	4848	33.400	33.000	33.300	.90	.30
761	5.100	5.000	5.000	0.00	1.96	4880	33.500	33.200	33.400	.60	.30
962	6.300	6.300	6.400	1.59	-1.59	4965	34.000	33.700	33.800	.29	.59
1100	7.300	7.200	7.200	0.00	1.37	4969	34.100	33.700	33.900	.59	.59
1380	9.100	9.000	9.000	0.00	1.10	5037	34.500	34.100	34.300	.58	.58
1446	9.600	9.400	9.500	1.04	1.04	5050	34.500	34.200	34.400	.58	.29
1622	10.600	10.600	10.600	0.00	0.00	5088	34.800	34.300	34.500	.57	.86
1878	12.300	12.100	12.300	1.63	0.00	5377	36.400	36.100	36.300	.55	.27
2003	13.200	13.000	13.000	0.00	1.52	5532	37.300	36.900	37.100	.54	.54
2013	13.200	13.100	13.200	.76	0.00	5636	37.900	37.400	37.700	.79	.53
2027	13.300	13.100	13.300	1.50	0.00	5735	38.400	38.000	38.200	.52	.52
2030	13.400	13.200	13.200	0.00	1.49	5850	39.100	38.700	39.000	.77	.26
2061	13.500	13.400	13.500	.74	0.00	5918	39.500	39.100	39.400	.76	.25
2085	13.700	13.500	13.600	.73	.73	6210	41.100	40.800	41.000	.49	.24
2126	14.000	13.800	13.900	.71	.71	6246	41.300	40.900	41.200	.73	.24
2263	14.800	14.700	14.700	0.00	.68	6270	41.500	41.100	41.300	.48	.48
2324	15.300	15.000	15.200	1.31	.65	6308	41.600	41.200	41.600	.96	0.00
2387	15.600	15.500	15.500	0.00	.64	6530	42.900	42.500	42.800	.70	.23
2397	15.600	15.500	15.600	.64	0.00	6623	43.500	43.000	43.300	.69	.46
2395	17.000	16.800	16.900	.59	.59	6748	44.100	43.700	44.100	.91	0.00
2604	17.000	16.900	17.000	.59	0.00	6755	44.300	43.800	44.000	.45	.68
2636	17.300	17.000	17.200	1.16	.58	6768	44.200	43.800	44.200	.90	0.00
2721	17.800	17.600	17.700	.56	.56	6986	45.500	45.100	45.300	.44	.44
2748	18.000	17.800	17.900	.56	.56	7051	45.900	45.400	45.800	.87	.22
2827	18.500	18.300	18.400	.54	.54	7091	46.100	45.600	46.000	.87	.22
3196	20.900	20.700	20.800	.48	.48	7111	46.200	45.800	46.100	.65	.22
3207	21.000	20.800	20.900	.48	.48	7139	46.400	45.900	46.300	.86	.22
3273	21.400	21.200	21.200	0.00	.93	7178	46.600	46.200	46.400	.43	.43
3292	21.500	21.300	21.400	.47	.47	7408	47.900	47.500	47.800	.63	.21
3321	21.700	21.500	21.600	.46	.46	7534	48.600	48.200	48.500	.62	.21
3339	21.700	21.500	21.700	.92	0.00	7607	49.000	48.600	48.900	.61	.20
3394	22.200	21.900	22.000	.45	.90	7799	50.100	49.600	50.000	.80	.20
3424	22.400	22.200	22.300	.45	.45	7813	50.200	49.700	50.100	.80	.20
3440	22.500	22.200	22.400	.89	.44	7822	50.300	49.800	50.200	.80	.20
3515	23.000	22.800	22.800	0.00	.87	7903	50.700	50.200	50.600	.79	.20
3554	23.200	22.900	23.100	.86	.43	7905	50.700	50.300	50.700	.79	0.00
3677	24.000	23.800	23.900	.42	.42	8108	53.700	53.200	53.600	.74	.19
3858	25.200	24.900	25.100	.79	.40	8146	53.900	53.300	53.800	.93	.19
3899	25.500	25.200	25.300	.39	.78	8190	54.900	54.600	54.800	.36	.18
3903	25.600	25.200	25.400	.78	.78	8192	55.000	54.500	54.900	.73	.18
3923	25.600	25.400	25.500	.39	.39						
3956	25.900	25.500	25.700	.77	.77						
4117	29.200	29.000	29.100	.34	.34						
4124	29.300	29.000	29.100	.68	.68						
4191	29.600	29.300	29.500	.68	.34						
4222	29.800	29.500	29.600	.34	.67						
						AVERAGE					
						SDO		PCO			
						.60		.54			

$$S_p = \frac{(i-1)P+k}{i} = P + (k-P)/i$$

Proceeding in a similar manner as version 1.0, we first average over all possible values for i , obtaining the average speed-up, $S_p(k)$ of the parallel algorithm, over all the keys found by processor k

$$S_p(k) = \frac{1}{N/P} \sum_{i=1}^{N/P} (P + (k-P)/i)$$

which simplifies to the following

$$S_p(k) = P + \left(\frac{k-P}{N/P}\right) \sum_{i=1}^{N/P} \frac{1}{i}$$

or if the sum term is bounded by $1 + \ln \frac{N}{P}$

$$S_p(k) < P + \left(\frac{k-P}{N/P}\right) \left(1 + \ln \frac{N}{P}\right)$$

The overall average speed of algorithm 2.0 is obtained by averaging $S_p(k)$ over all possible values of k .

$$S_p = \frac{1}{P} \sum_{k=1}^P S_p(k)$$

$$S_p < \frac{1}{P} \sum_{k=1}^P P + \frac{k-P}{N/P} \left(1 + \ln \frac{N}{P}\right) \quad (4.7)$$

which simplifies to the following expression after computing the corresponding sum:

$$S_p < P - \frac{P^2}{N} \left(1 + \ln \frac{N}{P}\right) + \frac{1}{N} \left(1 + \ln \frac{N}{P}\right) \sum_{k=1}^P k$$

or

$$\sum_{k=1}^P k = \frac{P(P+1)}{2}$$

Substituting the above sum in expression (4.7) we obtain

$$S_p < P - \frac{P^2}{N} \left(1 + \ln \frac{N}{P}\right) + \frac{P(P+1)}{2N} \left(1 + \ln \frac{N}{P}\right)$$

which is also

$$S_p < P \left[1 - \frac{(P-1)}{2N} \left(1 + \ln \frac{N}{P}\right)\right] \quad (4.8)$$

and

$$E_p = \frac{S_p}{P} = 1 - \frac{(P-1)}{2N} \left(1 + \ln \frac{N}{P}\right)$$

Note that as $N \rightarrow \infty$, S_p and E_p tend to P and 1 respectively.

In Table 4.9 we present the experimental average speed-ups of the parallel sequential search algorithm version 2.0, where each line corresponds to the performance of the considered algorithm when searching for a random key. A total of 100 such keys is searched and the experimental performance measurements of the algorithm are then computed.

The static overheads which are negligible are presented in Table 4.10.

4.3 A PARALLEL IMPLEMENTATION OF THE BINARY SEARCH

In the following section, we shall consider a multiprocessor implementation of a well known search algorithm, i.e. the binary search method, when searching an ordered set of records for the existence of a particular key. Based on the 'divide-and-conquer' strategy, the binary search algorithm proves to be very efficient when compared with the sequential algorithm. Using such a method,

TABLE 4.9: EXPERIMENTAL TIMING RESULTS OF THE PARALLEL SEQUENTIAL SEARCH ALGORITHM VERSION 2.0

T1	T2	T3	T4	T5	SP2	SP3	SP4	SP5	T1	T2	T3	T4	T5	SP2	SP3	SP4	SP5
.01	.03	.03	.02	.03	.33	.33	.50	.33	.77	.42	.29	.22	.19	1.83	2.66	3.50	4.05
.02	.04	.03	.03	.03	.50	.67	.67	.67	.78	.42	.30	.22	.18	1.86	2.60	3.55	4.33
.03	.03	.03	.04	.03	1.00	1.00	.75	1.00	.79	.41	.29	.22	.19	1.93	2.72	3.59	4.16
.05	.04	.04	.04	.04	1.25	1.25	1.25	1.25	.80	.43	.26	.23	.19	1.86	3.08	3.48	4.21
.10	.08	.06	.06	.05	1.25	1.67	1.67	2.00	.83	.43	.35	.23	.19	1.93	2.37	3.61	4.37
.10	.07	.05	.05	.04	1.43	2.00	2.00	2.50	.84	.45	.31	.23	.21	1.87	2.71	3.65	4.00
.11	.07	.07	.05	.05	1.57	1.57	2.20	2.20	.85	.46	.31	.28	.19	1.85	2.74	3.04	4.47
.11	.07	.06	.06	.06	1.57	1.83	1.83	1.83	.84	.45	.32	.24	.20	1.87	2.63	3.50	4.20
.13	.08	.07	.05	.05	1.63	1.86	2.60	2.60	.88	.48	.31	.28	.20	1.83	2.84	3.14	4.40
.12	.08	.07	.06	.05	1.50	1.71	2.00	2.40	.90	.47	.33	.27	.21	1.91	2.73	3.33	4.29
.15	.10	.08	.06	.05	1.50	1.88	2.50	3.00	.90	.48	.34	.25	.22	1.88	2.65	3.60	4.09
.16	.10	.08	.07	.06	1.60	2.00	2.29	2.67	.94	.49	.33	.26	.22	1.92	2.85	3.62	4.27
.16	.10	.08	.07	.05	1.60	2.00	2.29	3.20	.96	.52	.30	.26	.23	1.85	3.20	3.69	4.17
.15	.10	.07	.07	.06	1.50	2.14	2.14	2.50	.97	.49	.34	.28	.22	1.98	2.85	3.46	4.41
.19	.11	.09	.08	.07	1.73	2.11	2.38	2.71	.97	.53	.36	.28	.23	1.83	2.69	3.46	4.22
.19	.12	.08	.07	.06	1.58	2.38	2.71	3.17	1.00	.53	.35	.27	.23	1.89	2.86	3.70	4.35
.23	.13	.11	.09	.07	1.77	2.09	2.56	3.29	1.03	.56	.39	.16	.16	1.84	2.64	6.44	6.44
.24	.15	.10	.08	.08	1.60	2.40	3.00	3.00	1.01	.54	.39	.30	.23	1.87	2.59	3.37	4.39
.24	.14	.11	.10	.08	1.71	2.18	2.40	3.00	1.04	.55	.49	.29	.24	1.89	2.12	3.59	4.33
.30	.18	.12	.10	.10	1.67	2.50	3.00	3.00	1.07	.75	.50	.29	.26	1.43	2.14	3.69	4.12
.32	.18	.13	.11	.09	1.78	2.46	2.91	3.56	1.06	.58	.38	.29	.25	1.83	2.79	3.66	4.24
.33	.20	.14	.10	.09	1.65	2.36	3.30	3.67	1.09	.56	.40	.31	.46	1.95	2.73	3.52	2.37
.34	.19	.14	.12	.10	1.79	2.43	2.83	3.40	1.07	.43	.39	.31	.27	2.49	2.74	3.45	3.96
.39	.21	.15	.13	.12	1.86	2.60	3.00	3.25	1.11	.58	.40	.30	.26	1.91	2.78	3.70	4.27
.40	.23	.16	.13	.10	1.74	2.50	3.08	4.00	1.11	.59	.40	.30	.24	1.88	2.78	3.70	4.63
.47	.25	.17	.15	.12	1.88	2.76	3.13	3.92	1.13	.59	.40	.18	.17	1.92	2.83	6.28	6.65
.45	.25	.19	.15	.12	1.80	2.37	3.00	3.75	1.12	.63	.39	.27	.36	1.78	2.87	4.15	3.11
.48	.26	.18	.15	.13	1.85	2.67	3.20	3.69	1.16	.59	.41	.32	.26	1.97	2.83	3.63	4.46
.48	.27	.19	.14	.12	1.78	2.53	3.43	4.00	1.16	.61	.47	.32	.26	1.90	2.47	3.63	4.46
.53	.28	.20	.15	.15	1.89	2.65	3.53	3.53	1.17	.61	.41	.32	.28	1.92	2.85	3.66	4.18
.53	.29	.21	.16	.15	1.83	2.52	3.31	3.53	1.22	.64	.43	.33	.28	1.91	2.84	3.70	4.36
.52	.29	.21	.16	.15	1.79	2.48	3.25	3.47	1.20	.50	.43	.34	.27	2.40	2.79	3.53	4.44
.55	.31	.21	.19	.13	1.77	2.62	2.89	4.23	1.19	.70	.43	.33	.27	1.70	2.77	3.61	4.41
.56	.30	.11	.16	.14	1.87	5.09	3.50	4.00	1.25	.71	.50	.39	.29	1.76	2.50	3.21	4.31
.59	.33	.22	.17	.15	1.79	2.68	3.47	3.93	1.27	.83	.46	.37	.35	1.53	2.76	3.43	3.63
.60	.32	.20	.17	.15	1.88	3.00	3.53	4.00	1.25	.68	.46	.36	.39	1.84	2.72	3.47	3.21
.58	.32	.21	.17	.15	1.81	2.76	3.41	3.87	1.29	.69	.47	.37	.29	1.87	2.74	3.49	4.45
.60	.33	.22	.17	.15	1.82	2.73	3.53	4.00	1.32	.68	.46	.36	.34	1.94	2.87	3.67	3.88
.61	.33	.23	.18	.16	1.85	2.65	3.39	3.81	1.34	.71	.32	.37	.30	1.89	4.19	3.62	4.47
.64	.34	.23	.19	.16	1.88	2.78	3.37	4.00	1.33	.55	.49	.37	.29	2.42	2.71	3.59	4.59
.62	.35	.25	.19	.12	1.77	2.48	3.26	5.17	1.44	.76	.55	.38	.34	1.89	2.62	3.79	4.24
.64	.35	.24	.19	.14	1.83	2.67	3.37	4.57	1.43	.75	.54	.26	.37	1.91	2.65	5.50	3.86
.66	.36	.24	.19	.16	1.83	2.75	3.47	4.13	1.46	.80	.51	.47	.16	1.83	2.86	3.11	9.13
.67	.35	.25	.19	.16	1.91	2.68	3.53	4.19	1.45	.60	.52	.41	.38	2.42	2.79	3.54	3.82
.68	.36	.24	.20	.17	1.89	2.83	3.40	4.00	1.50	.78	.28	.41	.33	1.92	5.36	3.66	4.55
.69	.38	.21	.20	.18	1.82	3.29	3.45	3.83	1.52	.79	.55	.41	.34	1.92	2.76	3.71	4.47
.71	.38	.27	.21	.17	1.87	2.63	3.38	4.18	1.52	.85	.55	.45	.38	1.79	2.76	3.38	4.00
.74	.39	.25	.23	.17	1.90	2.96	3.22	4.35	1.53	.80	.54	.42	.45	1.91	2.83	3.64	3.40
.74	.40	.27	.21	.17	1.85	2.74	3.52	4.35	AVERAGE								
.77	.39	.27	.21	.18	1.97	2.85	3.67	4.28									
.74	.39	.28	.21	.18	1.90	2.64	3.52	4.11									
.78	.41	.28	.22	.18	1.90	2.79	3.55	4.33	.75	.40	.28	.22	.19	1.79	2.58	3.25	3.82

TABLE 4.10: THE MEASURED STATIC OVERHEADS OF THE PARALLEL SEQUENTIAL SEARCH ALGORITHM VERSION 2.0

IKEY	T1	TS	TN	SDO	PCO	IKEY	T1	TS	TN	SDO	PCO
125	.900	.800	.900	11.11	0.00	4222	29.200	29.200	29.700	.33	.67
124	1.700	1.500	1.500	0.00	11.76	4211	30.500	30.100	30.300	.36	0.60
100	2.100	2.100	2.100	0.00	0.00	4215	30.900	30.700	30.800	.32	.32
449	3.200	3.100	3.200	1.13	0.00	4217	31.200	31.500	31.700	.31	.31
483	3.300	3.100	3.200	-3.03	3.03	4260	32.000	31.700	31.800	.31	.61
541	4.500	4.500	4.400	-2.22	2.22	4271	32.400	32.400	32.500	.31	.31
691	4.600	4.500	4.600	2.17	0.00	4219	32.700	32.400	32.500	.31	.61
739	4.900	4.800	4.900	2.04	0.00	4275	33.400	33.100	33.200	.30	.60
741	5.100	5.000	5.000	0.00	1.96	4249	33.500	33.200	33.300	.30	.60
842	6.400	6.100	6.400	1.56	0.00	4800	33.700	33.300	33.500	.29	.59
1100	7.300	7.300	7.300	0.00	0.00	4945	34.100	33.800	33.900	.29	.29
1390	9.100	9.100	9.100	0.00	0.00	4969	34.200	33.800	33.900	.29	.88
1446	9.600	9.400	9.500	1.04	1.04	5077	34.500	34.200	34.400	.28	.29
1632	10.700	10.600	10.600	0.00	.93	5050	34.600	34.300	34.500	.28	.29
1878	12.400	12.300	12.400	.81	0.00	5099	34.800	34.500	34.600	.29	.57
2003	13.200	13.000	13.100	.76	.76	5377	36.500	36.200	36.300	.27	.55
2011	13.300	13.200	13.200	0.00	.73	5532	37.400	37.000	37.200	.26	.53
2021	13.400	13.200	13.300	.75	.75	5636	38.000	37.700	37.800	.26	.53
2030	13.400	13.200	13.300	.75	.75	5735	38.500	38.100	38.300	.26	.52
2041	13.600	13.400	13.500	.74	.74	5850	39.200	38.900	39.000	.26	.51
2085	13.700	13.600	13.700	.73	0.00	5918	39.500	39.200	39.400	.26	.25
2126	14.000	13.900	14.000	.71	0.00	6210	41.200	40.900	41.000	.24	.49
2263	14.900	14.800	14.800	0.00	.67	6246	41.500	41.200	41.300	.24	.48
2324	15.300	15.200	15.200	0.00	.63	6270	41.600	41.300	41.400	.24	.48
2387	15.800	15.500	15.700	1.27	.63	6308	41.900	41.400	41.500	.24	.93
2397	15.800	15.600	15.700	.63	.63	6530	43.100	42.800	42.900	.23	.46
2593	17.000	16.900	17.000	.59	0.00	6623	43.600	43.200	43.400	.46	.46
2604	17.100	16.900	17.100	1.17	0.00	6749	44.400	44.000	44.100	.23	.68
2636	17.300	17.200	17.200	0.00	.58	6755	44.400	44.000	44.100	.23	.68
2721	18.000	17.800	17.800	1.11	1.11	6768	44.400	44.000	44.200	.45	.45
2748	18.000	17.900	17.900	0.00	.36	6986	45.700	45.300	45.400	.22	.66
2827	18.600	18.400	19.500	.54	.54	7051	46.100	45.700	45.800	.22	.65
3106	21.000	20.800	20.900	.48	.48	7091	46.200	45.900	46.100	.43	.22
3207	21.100	20.900	20.900	0.00	.95	7111	46.400	46.100	46.100	0.00	.65
3273	21.500	21.300	21.400	.47	.47	7139	46.500	46.100	46.300	.43	.43
3282	21.600	21.400	21.500	.46	.46	7178	46.800	46.400	46.500	.21	.64
3301	21.800	21.600	21.700	.46	.46	7408	48.100	47.700	47.900	.42	.42
3350	21.800	21.600	21.700	.46	.46	7534	48.200	48.400	48.500	.20	.61
3374	22.300	22.000	22.200	.90	.45	7607	49.300	48.800	49.000	.41	.61
3424	22.300	22.300	22.300	0.00	.89	7799	50.300	50.000	50.100	.20	.40
3440	22.300	22.300	22.300	.44	.88	7917	50.400	50.000	50.100	.20	.60
3451	22.300	22.300	22.300	.43	.43	7922	50.400	50.000	50.200	.40	.40
3459	22.300	22.300	22.300	.43	.86	7907	50.900	50.500	50.600	.20	.59
3461	22.300	22.300	22.300	.41	.83	7907	51.000	50.500	50.600	.20	.78
3468	22.300	22.300	22.300	.40	.40	8109	53.900	53.500	53.600	.19	.56
3469	22.300	22.300	22.300	.39	.78	8146	54.100	53.700	53.700	0.00	.74
3469	22.300	22.300	22.300	.78	.39	8190	55.300	54.800	55.000	.36	.54
3469	22.300	22.300	22.300	.78	0.00	8192	55.300	54.800	55.000	.36	.54
3469	22.300	22.300	22.300	.32	.77	AVERAGE					
3469	22.300	22.300	22.300	.34	.34						SDO PCO
3469	22.300	22.300	22.300	.34	.34						.51 .66

the total search time can be considerably reduced to $\log(N)+1$, where N is the size of the set.

Basically, the binary search divides the set into two subsets and compares the key with the element at the middle position of the set. From the result of this comparison, it is possible to determine which of the two subsets the key being sought belongs to, then concentrates only on that half. Consequently, the interval size is at least halved at each iteration, so the total search time complexity is proportional to $\log(N)+1$, where N is the size of the original set of records.

The sequence of key comparisons made by the binary search algorithm is predetermined. More specifically, it is based on the value of the key being sought and the value of N . Thus, if a structure is to be used to represent all these decisions, one would choose a binary tree structure. For example, to search a telephone directory, for a name starting with S the following binary tree describes the comparison structure of a possible search.

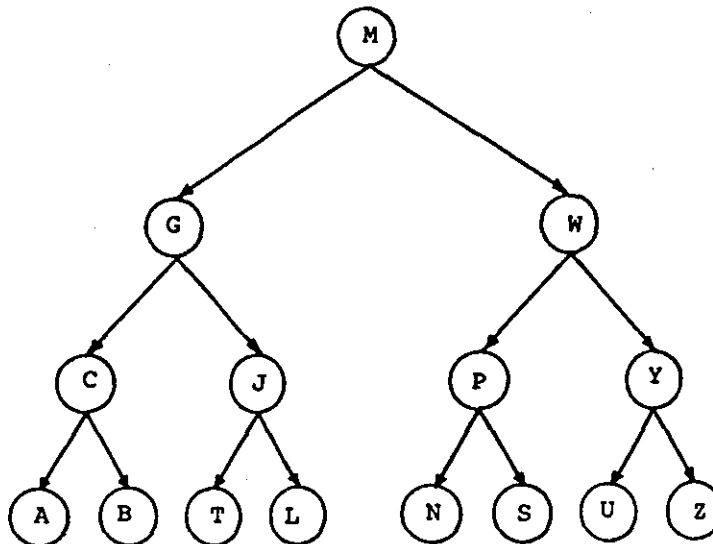


FIGURE 4.3: A BINARY TREE REPRESENTATION OF THE COMPARISON DECISIONS

For the parallel implementation of the binary search algorithm we suggested and analysed three different versions of parallel algorithms based on the partition of the original set of records among the P available processors. The first version allocates to each processor k , $k = 1, 2, \dots, P$ the records stored at the following locations:

$$(k-1) \frac{N}{P} + 1$$

where $i = 1, 2, \dots, \frac{N}{P}$. The disadvantage of such a version lies in the fact that all the processors will soon become idle (and that after a single key comparison) while only one is searching the subset which is likely to contain the target key because all, except one, found that the key is outside their subset and there is no reason to carry on searching. The second version which allocates to each processor k , $k = 1, 2, \dots, P$ the following locations:

$$(i-1) P + k$$

where $i = 1, 2, \dots, \frac{N}{P}$ removes the above anomaly and consequently keeps all the processors busy until the key is found by one of them. However, only one processor is performing useful work (search) since we know that the other searches are failures. In the last version, we approached the problem differently from the two previous ones. In the two first versions, once the set is partitioned, each processor will perform the binary search algorithm until the key is either found or not. However with version 3.0, each processor performs a single iteration of the binary search. If the key is found, then the algorithm terminates successfully, otherwise one of the active processors defines the bound location indexes of the new subset that is likely to contain the desired key. In the case that

the key is not yet found, this new subset is then partitioned into P smaller subsets and the process continues until the key is found. Consequently, this version has the advantage that the size of the set to be searched in the next iteration decreases much faster than in the other two versions. In versions 1.0 and 2.0, the subset is halved into two portions in every iteration whereas it is divided by $2P$ in the third version.

The analysis of each version, as well as experimental results obtained when implementing the algorithms on the Neptune system, is discussed in the following sub-sections:

4.3.1 PARALLEL BINARY SEARCH VERSIONS 1.0 AND 2.0

We assume, for simplicity in the analysis, that N , the size of the set, and P are powers of 2. A model for the parallel binary search on an MIMD multiprocessor with negligible overheads is as follows.

Using a single processor, it takes at least $\log N + 1$ key comparisons in the worst case to determine whether or not a given key exists.

With p processors in use, where each of which applies the binary algorithm to search a subset of $\frac{N}{P}$ elements, we require, in the worst case, $\log \frac{N}{P} + 1$ key comparisons. Thus, the speed-up in the worst case is

$$S_p = \frac{T}{T_p} \leq \frac{\log N + 1}{\log \frac{N}{P} + 1}$$

$$S_p \leq 1 + \frac{\log P}{\log \frac{N}{P} + 1}$$

$$E_p = \frac{S_p}{P}$$

TABLE 4.12: THE EXPERIMENTAL RESULTS OF THE PARALLEL
BINARY SEARCH ALGORITHM VERSION 1.0

IKEY	T1	T2	T3	T5	SP2	SP3	SP4	IKEY	T1	T2	T3	T5	SP2	SP3	SP4
107	20.600	19.300	17.500	18.000	1.067	1.177	1.144	4121	21.200	30.100	26.000	29.400	.704	.815	.721
122	20.100	18.700	18.600	17.400	1.075	1.081	1.155	4194	20.600	30.300	31.000	29.700	.680	.665	.718
376	17.100	15.800	19.300	15.300	1.082	.886	1.118	4205	21.300	30.300	31.400	29.600	.703	.678	.720
463	20.500	19.200	18.800	17.900	1.068	1.090	1.145	4223	20.500	30.200	29.100	28.900	.679	.704	.709
534	20.000	18.600	17.100	17.200	1.075	1.170	1.163	4233	21.300	30.300	29.600	29.200	.703	.720	.729
610	19.900	18.600	19.300	17.300	1.070	1.031	1.150	4280	17.500	25.700	26.100	25.300	.681	.670	.692
675	20.700	19.300	19.600	18.000	1.073	1.056	1.150	4409	21.100	30.600	30.300	29.400	.690	.694	.718
754	20.100	18.700	18.900	17.400	1.075	1.063	1.155	4558	20.600	30.300	25.800	28.600	.680	.798	.720
783	20.600	19.300	18.700	18.100	1.067	1.102	1.138	4579	21.100	29.900	26.300	29.900	.706	.802	.706
799	20.400	18.900	17.900	17.800	1.079	1.140	1.146	4605	21.300	30.100	30.300	29.500	.708	.703	.722
1097	20.600	19.200	15.500	18.000	1.073	1.329	1.144	4741	21.200	31.500	29.200	29.900	.673	.726	.709
1108	18.300	17.100	17.500	16.100	1.070	1.046	1.137	4776	17.200	25.700	30.700	25.100	.669	.560	.685
1170	20.000	18.600	18.700	17.200	1.075	1.070	1.163	4800	12.500	22.600	28.700	20.600	.553	.436	.607
1310	19.800	18.500	19.200	17.200	1.070	1.031	1.151	4802	20.700	30.500	29.900	28.800	.679	.692	.719
1412	18.300	17.000	15.600	15.900	1.076	1.173	1.151	4845	21.100	30.100	30.400	29.200	.701	.694	.723
1419	20.600	19.200	17.000	17.900	1.073	1.212	1.151	4882	20.500	30.100	30.600	28.500	.681	.670	.719
1466	20.000	18.600	18.600	17.200	1.075	1.075	1.163	4903	21.100	30.800	30.000	29.500	.685	.703	.715
1490	19.800	18.600	18.700	17.300	1.065	1.059	1.145	4947	21.200	30.300	30.100	29.400	.700	.704	.721
1512	16.900	15.600	18.900	14.700	1.083	.894	1.150	5162	20.500	30.500	28.200	28.400	.672	.727	.722
1523	20.600	19.200	19.100	17.900	1.073	1.079	1.151	5183	20.500	30.800	28.400	29.000	.664	.722	.707
1611	20.500	19.100	17.200	17.800	1.073	1.192	1.152	5206	20.600	30.000	29.000	28.600	.687	.710	.720
1633	20.300	18.900	18.700	17.600	1.074	1.086	1.153	5289	21.100	30.200	30.100	29.400	.699	.701	.718
1655	20.600	19.100	18.700	18.000	1.079	1.102	1.144	5386	20.600	30.700	29.500	28.600	.671	.698	.720
1793	20.000	18.700	17.000	17.200	1.070	1.176	1.163	5394	20.500	30.000	30.100	28.500	.683	.681	.719
1829	20.600	19.300	18.500	17.800	1.067	1.114	1.157	5485	21.100	31.200	30.600	29.400	.676	.690	.718
1958	20.000	18.600	19.400	17.200	1.075	1.031	1.163	5525	21.200	30.900	28.600	29.800	.686	.741	.711
1980	18.400	17.100	16.900	16.000	1.076	1.089	1.150	5553	21.000	30.500	30.700	29.400	.689	.684	.714
2079	20.200	18.900	16.400	28.400	1.069	1.232	.711	5726	20.400	29.300	29.100	28.600	.696	.701	.713
2119	20.500	19.200	17.100	28.600	1.068	1.199	.717	5923	21.400	31.200	26.200	29.100	.686	.817	.735
2201	20.400	19.200	15.700	28.600	1.062	1.299	.713	6292	18.900	28.300	29.100	26.000	.668	.649	.727
2234	19.900	18.700	18.600	27.900	1.064	1.070	.713	6371	21.200	30.700	29.900	29.100	.691	.709	.729
2363	20.500	19.200	19.300	28.800	1.068	1.062	.712	6439	21.100	31.300	30.100	28.900	.674	.701	.730
2454	19.800	18.500	17.100	28.100	1.070	1.138	.705	6744	17.300	25.500	30.800	25.600	.678	.562	.676
2536	16.700	15.500	18.800	25.000	1.077	.888	.668	6798	20.400	30.000	25.700	28.400	.680	.794	.718
2563	20.700	19.500	18.800	29.300	1.062	1.101	.706	6888	17.300	25.700	30.000	25.500	.673	.577	.678
2632	16.700	15.700	18.700	25.300	1.064	.893	.660	7034	20.600	30.600	30.500	28.300	.673	.673	.728
2933	20.600	19.200	27.600	29.200	1.073	.746	.705	7161	21.300	30.900	25.800	29.100	.689	.824	.732
2998	19.900	18.700	29.100	27.900	1.064	.684	.713	7240	17.300	25.600	30.500	25.800	.676	.567	.671
3021	20.600	19.200	29.800	28.700	1.073	.691	.718	7341	21.300	31.100	29.900	29.200	.685	.712	.729
3089	20.500	19.400	27.100	28.800	1.057	.756	.712	7345	20.900	30.500	28.200	28.500	.685	.741	.733
3092	18.300	17.100	22.900	25.800	1.070	.799	.709	7375	20.900	31.200	29.700	28.700	.670	.704	.728
3096	16.800	15.800	29.600	24.900	1.063	.568	.675	7388	18.800	28.400	29.900	26.000	.662	.629	.723
3141	20.600	19.300	29.800	28.600	1.067	.691	.720	7512	17.300	25.600	30.800	25.500	.676	.562	.678
3148	18.300	17.200	30.000	25.800	1.064	.610	.709	7804	18.900	28.100	30.900	25.700	.673	.612	.735
3151	20.300	19.100	29.700	28.600	1.063	.684	.710	7855	20.800	30.600	30.000	28.700	.680	.693	.725
3250	19.900	18.800	29.700	28.200	1.059	.670	.706	7985	21.100	30.200	29.500	29.200	.699	.715	.723
3282	19.800	18.700	29.800	27.900	1.059	.664	.710	8028	19.000	28.500	28.900	25.800	.667	.657	.736
3324	18.500	17.100	29.400	25.600	1.082	.629	.723	8155	21.400	30.700	25.900	29.300	.697	.826	.730
3364	18.200	17.100	29.900	25.500	1.064	.609	.714	AVERAGE							
3385	20.400	19.200	29.200	28.500	1.062	.699	.716	19.869	23.718	25.071	25.050	.893	.837	.829	
3400	16.800	15.500	29.900	25.200	1.084	.562	.667								
3838	20.000	19.000	20.100	28.800	1.053	.995	.694								

TABLE 4.13: THE MEASURED STATIC OVERHEADS OF THE
PARALLEL BINARY SEARCH ALGORITHM VERSION 1.0

IKEY	T1	TS	TN	SDO	PCO	IKEY	T1	TS	TN	SDO	PCO
107	20.600	18.300	18.400	.49	10.68	3838	20.100	17.800	17.800	0.00	11.44
122	20.000	17.800	17.800	0.00	11.00	4121	21.100	18.900	18.900	0.00	10.43
376	17.100	14.700	14.800	.58	13.45	4194	20.700	18.300	18.400	.48	11.11
463	20.500	18.200	18.300	.49	10.73	4205	21.200	18.900	19.000	.47	10.38
534	19.900	17.600	17.700	.50	11.06	4223	20.500	18.200	18.200	0.00	11.22
610	19.900	17.600	17.600	0.00	11.56	4233	21.300	19.000	19.000	0.00	10.80
673	20.700	18.400	18.500	.48	10.63	4280	17.400	15.100	15.100	0.00	13.22
754	20.100	17.900	17.800	-.50	11.44	4409	21.100	18.900	18.900	0.00	10.43
783	20.600	18.300	18.400	.49	10.68	4558	20.600	18.300	18.300	0.00	11.17
799	20.300	18.000	18.000	0.00	11.33	4579	21.200	18.800	18.900	.47	10.85
1097	20.500	18.200	18.400	.98	10.24	4605	21.300	19.100	19.100	0.00	10.33
1108	18.400	16.100	16.100	0.00	12.50	4741	21.200	18.800	19.000	.94	10.38
1170	20.000	17.600	17.700	.50	11.50	4776	17.200	14.900	14.900	0.00	13.37
1310	19.900	17.600	17.700	.50	11.06	4800	12.600	10.300	10.300	0.00	18.25
1412	18.400	16.100	16.100	0.00	12.50	4802	20.600	18.300	18.400	.49	10.68
1419	20.500	18.200	18.300	.49	10.73	4845	21.100	18.800	18.900	.47	10.43
1466	19.900	17.600	17.700	.50	11.06	4882	20.400	18.100	18.200	.49	10.78
1490	19.900	17.500	17.600	.50	11.56	4903	21.100	18.800	18.900	.47	10.43
1512	16.800	14.500	14.600	.60	13.10	4947	21.100	18.900	18.900	0.00	10.43
1523	20.600	18.400	18.400	0.00	10.68	5162	20.500	18.200	18.300	.49	10.73
1611	20.500	18.300	18.400	.49	10.24	5183	20.500	18.300	18.300	0.00	10.73
1633	20.200	17.900	18.100	.99	10.40	5206	20.500	18.200	18.300	.49	10.73
1653	20.600	18.300	18.200	-.49	11.65	5289	21.100	18.800	18.800	0.00	10.90
1793	20.100	17.800	17.800	0.00	11.44	5386	20.600	18.300	18.300	0.00	11.17
1829	20.500	18.200	18.300	.49	10.73	5394	20.500	18.200	18.200	0.00	11.22
1958	20.000	17.700	17.700	0.00	11.50	5485	21.200	18.800	19.000	.94	10.38
1980	18.400	16.100	16.100	0.00	12.50	5525	21.200	18.900	19.000	.47	10.38
2079	20.200	17.800	18.000	.99	10.89	5553	21.000	18.600	18.800	.95	10.48
2119	20.500	18.200	18.200	0.00	11.22	5726	20.300	18.100	18.200	.49	10.34
2201	20.400	18.100	18.200	.49	10.78	5923	21.200	19.000	19.000	0.00	10.38
2234	19.800	17.600	17.600	0.00	11.11	6292	18.900	16.600	16.700	.53	11.64
2363	20.400	18.200	18.200	0.00	10.78	6371	21.200	18.900	19.000	.47	10.38
2454	19.800	17.500	17.500	0.00	11.62	6439	21.000	18.600	18.700	.48	10.95
2536	16.700	14.400	14.400	0.00	13.77	6744	17.300	15.000	15.000	0.00	13.29
2543	20.700	18.300	18.500	.97	10.63	6798	20.500	18.100	18.200	.49	11.22
2670	16.700	14.500	14.500	-.60	13.77	6888	17.100	14.900	15.000	.58	12.28
2931	20.500	18.200	18.300	0.00	10.73	7034	20.500	18.300	18.300	0.00	10.73
2998	19.800	17.600	17.600	0.00	11.11	7161	21.200	19.000	19.100	.47	9.91
3021	20.500	18.300	18.200	0.00	10.73	7240	17.300	15.000	15.000	0.00	13.29
3089	20.500	18.300	18.200	0.00	10.73	7341	21.200	18.900	19.000	.47	10.38
3092	18.300	16.000	16.000	0.00	12.57	7345	20.800	18.500	18.700	.94	10.10
3096	16.700	14.500	14.500	0.00	13.17	7375	20.800	18.600	18.700	.48	10.10
3141	20.500	18.200	18.200	.99	11.22	7388	18.900	16.500	16.600	.53	12.17
3148	18.300	16.000	16.100	1.09	12.02	7512	17.200	15.000	15.000	0.00	12.79
3151	20.200	18.000	18.000	0.00	10.89	7804	18.900	16.500	16.600	.53	12.17
3250	20.000	17.700	17.700	0.00	11.50	7855	20.800	18.500	18.500	0.00	11.03
3282	19.900	17.500	17.600	.50	11.56	7985	21.000	18.800	18.900	.48	10.60
3324	18.300	16.000	16.100	.55	12.02	8028	19.000	16.700	16.700	0.00	12.11
3364	18.300	16.000	16.000	.55	12.57	8155	21.400	19.000	19.100	.47	10.75
3383	20.300	18.000	18.100	.49	10.84						
3400	16.800	14.400	14.500	.60	13.69					.20	11.75

Table 4.11 shows the theoretical speed-up S_p and efficiency E_p versus P when $N = 8192$ records.

P	2	4	8	16
S_p	1.077	1.167	1.273	1.400
E_p	0.538	0.292	0.159	0.087

TABLE 4.11: EXPECTED THEORETICAL SPEED-UP AND EFFICIENCY OF THE PARALLEL BINARY SEARCH

Experiments on the Neptune system have shown that for both versions of the parallel binary search, the average speed S_p , $p = 2, 3, 4$ is less than unity, see Tables 4.12 and 4.14.

From both theoretical and experimental points of view, the parallel binary search versions 1.0 and 2.0 achieve poor average performance results. Theoretically though (see Table 4.11), there is a slight increase in speed as the number of processors is doubled.

The static overheads for both methods are negligible (see Tables 4.13 and 4.15).

The reason for such a poor performance for these parallel algorithms lies in the fact that adding more processors does not equally split the total task amongst the processors. For instance, if we double the number of processors from P to $2P$ we reduce T_p by only a single key comparison, since $\log \frac{N}{2P} = \log \frac{N}{P} - 1$. Version 3.0, which shall be analysed in the next paragraph, is likely to improve this poor performance since it manages to split the jobs equally amongst the processors.

TABLE 4.14: THE EXPERIMENTAL RESULTS OF THE PARALLEL BINARY
SEARCH ALGORITHM VERSION 2.0

KEY	T1	T2	T4	SP2	SP4	KEY	T1	T2	T4	SP2	SP4
107	17.000	25.800	27.100	.659	.627	4121	17.500	26.700	28.100	.655	.623
122	16.900	26.300	27.100	.643	.624	4194	17.300	26.600	27.800	.650	.622
376	14.500	24.700	27.100	.587	.535	4205	17.400	27.000	25.200	.644	.690
463	17.100	26.600	26.400	.643	.648	4223	17.400	26.600	26.800	.654	.649
534	17.000	26.400	27.000	.644	.630	4233	17.600	27.600	28.400	.638	.620
610	16.900	26.200	26.500	.645	.638	4280	14.800	25.300	27.000	.585	.548
675	17.200	27.000	26.300	.637	.654	4409	17.500	27.000	27.700	.648	.632
754	17.000	26.600	26.600	.639	.639	4558	17.300	26.700	27.700	.648	.625
783	17.100	26.700	26.500	.640	.645	4579	17.500	26.400	27.500	.663	.636
799	17.100	26.200	26.000	.653	.658	4605	17.300	27.000	29.100	.641	.595
1097	17.100	26.300	27.200	.650	.629	4741	17.500	27.200	27.000	.643	.648
1108	15.800	26.400	26.900	.598	.587	4774	14.800	25.100	27.200	.590	.544
1170	17.000	26.500	26.800	.642	.634	4800	11.100	22.200	27.200	.500	.408
1310	17.000	26.000	27.100	.654	.627	4802	17.200	26.900	27.600	.639	.623
1412	15.700	26.100	27.100	.602	.579	4845	17.400	27.000	26.900	.644	.647
1419	17.100	27.200	27.500	.629	.622	4882	17.300	26.900	27.600	.643	.627
1466	17.000	26.600	27.000	.639	.630	4903	17.500	26.100	27.100	.670	.646
1490	16.900	27.200	27.300	.621	.619	4947	17.500	27.200	27.700	.643	.632
1512	14.600	25.000	26.700	.584	.547	5162	17.300	27.400	27.400	.631	.631
1523	17.000	26.200	26.800	.649	.634	5183	17.500	26.800	26.800	.653	.653
1611	17.100	26.100	27.000	.655	.633	5206	17.300	27.100	27.500	.638	.629
1633	17.100	26.100	26.900	.655	.636	5289	17.500	26.900	27.100	.651	.646
1655	17.200	25.700	27.000	.649	.637	5386	17.300	26.700	27.200	.648	.636
1793	17.100	26.500	26.700	.645	.640	5394	17.300	26.900	27.200	.643	.636
1829	17.000	26.800	26.800	.634	.634	5485	17.500	27.100	26.100	.646	.670
1958	17.000	26.500	27.100	.642	.627	5525	17.500	26.800	27.800	.653	.629
1980	15.600	26.100	26.800	.598	.582	5553	17.500	26.700	26.900	.655	.651
2119	17.100	25.800	26.800	.663	.638	5726	17.200	27.100	27.200	.635	.632
2201	17.200	26.100	26.700	.659	.644	5923	17.600	27.100	28.000	.649	.629
2234	16.900	26.400	26.400	.640	.640	6292	16.000	27.100	27.300	.590	.586
2363	17.100	27.000	27.400	.633	.624	6371	17.500	27.300	27.600	.641	.634
2454	17.000	26.600	26.700	.639	.637	6439	17.500	26.000	27.000	.673	.648
2536	14.400	25.300	26.800	.569	.537	6744	14.800	25.300	27.700	.585	.534
2563	17.100	26.800	26.700	.638	.640	6798	17.200	26.900	26.800	.639	.642
2632	14.400	24.900	26.800	.578	.537	6888	14.800	25.100	27.100	.590	.546
2933	17.000	26.500	26.400	.642	.644	7034	17.200	27.200	27.100	.632	.635
2998	16.900	26.700	27.000	.633	.626	7161	17.500	27.200	28.300	.643	.618
3021	17.000	26.000	26.200	.654	.649	7240	14.800	25.200	27.200	.587	.544
3089	17.100	26.700	26.500	.640	.645	7341	17.500	27.300	26.200	.641	.668
3092	15.700	26.100	26.500	.602	.592	7345	17.500	27.000	27.300	.648	.641
3096	14.500	25.100	26.800	.578	.541	7375	17.400	26.700	26.900	.652	.647
3141	17.100	26.700	26.400	.640	.648	7388	16.000	26.400	27.700	.606	.578
3148	15.700	26.200	27.200	.599	.577	7512	14.800	25.400	26.900	.583	.550
3151	17.100	26.500	26.000	.645	.658	7804	15.900	26.800	27.100	.593	.587
3250	17.000	26.300	27.100	.646	.627	7855	17.400	27.400	26.900	.635	.647
3282	16.900	26.100	26.400	.648	.635	7985	17.400	27.000	27.000	.644	.644
3324	15.700	26.500	26.800	.592	.586	8028	16.000	26.900	28.100	.595	.569
3364	15.700	26.400	26.700	.595	.588	8155	17.500	26.600	26.900	.658	.651
3385	17.100	26.700	27.400	.640	.624	AVERAGE					
3400	14.500	25.000	26.300	.580	.551						
3838	16.900	26.100	26.500	.648	.638		15.695	26.414	27.025	.632	.618

TABLE 4.15: THE MEASURED STATIC OVERHEADS OF THE PARALLEL
BINARY SEARCH ALGORITHM VERSION 2.0

IKEY	T1	TS	TN	SDO	PCO
107	17.000	14.700	14.800	.59	12.94
122	17.000	14.600	14.600	0.00	14.12
376	14.500	12.200	12.200	0.00	15.86
463	17.100	14.800	14.900	.58	12.87
534	16.900	14.600	14.700	.59	13.02
610	16.900	14.700	14.700	0.00	13.02
675	17.100	14.800	14.800	0.00	13.45
754	16.900	14.600	14.700	.59	13.02
783	17.100	14.800	14.800	0.00	13.45
799	17.000	14.800	14.800	0.00	12.94
1097	17.200	14.900	14.900	0.00	13.37
1108	15.800	13.400	13.500	.63	14.56
1170	17.000	14.700	14.700	0.00	13.53
1310	16.900	14.700	14.800	.59	12.43
1412	15.700	13.500	13.400	-.64	14.65
1419	17.100	14.800	14.800	0.00	13.45
1466	16.900	14.600	14.700	.59	13.02
1490	16.900	14.700	14.600	-.59	13.61
1512	14.500	12.300	12.300	0.00	15.17
1524	17.100	14.700	14.700	0.00	14.04
1611	17.000	14.700	14.800	.59	12.94
1633	17.200	14.900	14.800	-.58	13.95
1653	17.200	14.800	14.900	.58	13.37
1793	17.100	14.800	14.800	0.00	13.45
1829	17.100	14.700	14.800	.58	13.45
1958	16.900	14.700	14.700	0.00	13.02
1980	15.700	13.300	13.400	.64	14.65
2079	17.100	14.800	14.800	0.00	13.45
2119	17.200	14.900	14.900	0.00	13.37
2201	17.100	14.800	14.800	0.00	13.45
2234	16.900	14.600	14.700	.59	13.02
2363	17.200	14.800	14.800	0.00	13.95
2454	16.900	14.600	14.700	.59	13.02
2536	14.500	12.300	12.200	0.00	15.86
2563	17.200	14.900	15.000	.58	12.79
2632	14.600	12.200	12.200	0.00	16.44
2933	17.000	14.800	14.800	0.00	12.94
2998	17.000	14.600	14.600	0.00	14.12
3021	17.100	14.800	14.800	0.00	13.45
3089	17.100	14.800	14.900	.58	12.87
3092	15.800	13.300	13.400	-.63	15.19
3096	14.500	12.200	12.300	.69	15.17
3141	17.100	14.800	14.800	0.00	13.45
3151	17.100	14.800	14.800	0.00	13.45
3250	17.000	14.700	14.700	0.00	13.53
3282	16.900	14.600	14.600	0.00	13.61
3324	15.700	13.400	13.500	.64	14.01
3364	15.800	13.400	13.500	.63	14.56
3383	17.100	14.800	14.800	0.00	13.45
3400	14.500	12.200	12.300	.69	15.17
3838	16.900	14.600	14.700	.59	13.02
4121	17.600	15.200	15.200	0.00	13.64
4194	17.300	15.000	15.100	.58	12.72
4205	17.400	15.100	15.200	.57	12.64
4223	17.400	15.100	15.200	.57	12.64

IKEY	T1	TS	TN	SDO	PCO
4233	17.500	15.200	15.200	0.00	13.14
4280	14.800	12.600	12.600	0.00	14.86
4409	17.500	15.300	15.300	0.00	12.57
4558	17.200	15.000	15.000	0.00	12.79
4579	17.400	15.200	15.200	0.00	12.64
4605	17.400	15.100	15.200	.57	12.64
4741	17.500	15.100	15.200	.57	13.14
4776	14.800	12.500	12.500	0.00	15.54
4800	11.100	8.800	8.900	.90	19.82
4802	17.200	15.000	15.000	0.00	12.79
4845	17.500	15.100	15.200	.57	13.14
4882	17.300	15.000	15.100	.58	12.72
4903	17.500	15.200	15.300	.57	12.57
4947	17.500	15.200	15.200	0.00	13.14
5162	17.300	15.000	15.000	0.00	13.29
5183	17.400	15.100	15.200	.57	12.64
5206	17.300	15.000	15.000	0.00	13.29
5289	17.500	15.200	15.200	0.00	13.14
5386	17.300	15.000	15.000	0.00	13.29
5394	17.300	14.900	15.000	.58	13.29
5485	17.500	15.200	15.200	0.00	13.14
5525	17.500	15.200	15.300	.57	12.57
5553	17.500	15.200	15.200	0.00	13.14
5726	17.200	14.900	14.900	0.00	13.37
5923	17.600	15.300	15.300	0.00	13.07
6292	16.000	13.700	13.800	.63	13.75
6371	17.500	15.200	15.200	0.00	13.14
6439	17.400	15.200	15.100	-.57	13.22
6744	14.800	12.500	12.600	.68	14.86
6798	17.300	14.900	14.900	0.00	13.87
6888	14.800	12.500	12.500	0.00	15.54
7034	17.200	14.900	15.000	.58	12.79
7161	17.400	15.200	15.200	0.00	12.64
7240	14.800	12.500	12.500	0.00	15.54
7341	17.400	15.200	15.200	0.00	12.64
7345	17.400	15.100	15.200	.57	12.64
7375	17.400	15.100	15.200	.57	12.64
7388	16.000	13.700	13.700	0.00	14.38
7512	14.800	12.500	12.500	0.00	15.54
7804	16.000	13.800	13.700	-.63	14.38
7855	17.500	15.100	15.200	.57	13.14
7985	17.400	15.200	15.100	-.57	13.22
8028	16.000	13.600	13.700	.63	14.38
8155	17.400	15.200	15.200	0.00	12.64

AVERAGES OF THE SDO AND PCO.					
				AVER.SDO	AVER.PCO
				.20	13.63

4.3.2 PARALLEL BINARY SEARCH VERSION 3.0

We noted that one of the reasons for the poor performance of the parallel binary search versions 1.0 and 2.0, is that only one processor is kept busy, doing effective work, after the first iteration. In particular for version 1.0, all the processors except one, soon discover that the target key is outside the subset they were allocated to and there is no need to carry on searching. A third method was suggested to remedy this inconvenience and consequently it is most likely to improve, at least, the theoretical performance.

At the end of every iteration where P key locations are compared with the key, all the processors are forced to a synchronisation point. When all have joined this point, a single processor is used to define the bounds of the new subset. Then the process is repeated until the key is found or the set is fully exhausted.

For the analysis of this parallel version, we proceed as follows. Using a single processor, we make at least $\log N+1$ key comparisons, in the worst case

$$T_1 \leq \log N+1$$

With P processors, the subset size is divided by $2P$ in every iteration. Only one of these $2P$ new subsets is considered in the following iteration. So, in the worst case:

$$T_p = \log_{2P} N+1$$

Therefore, the speed-up is obtained as:

$$S_p = \frac{T_1}{T_p} \leq \frac{\log N + 1}{\log_{2P}^N + 1}$$

or

$$\begin{aligned} \log_{2P}^N + 1 &> \log_{2P} N \\ &= \frac{\log N}{\log 2P} \end{aligned}$$

Thus

$$S_p \leq \frac{\log N + 1}{\log N} * (\log P + 1)$$

$$S_p \leq \log P + 1$$

and

$$E_p = \frac{S_p}{P} \leq \frac{\log P + 1}{P}$$

From this simple model, we have shown that the speed-up of version 3.0 is logarithmic in the number of processors. However, with a more realistic model which includes some overheads (in particular, synchronisation overheads), it is expected that the experimental speed-up is no more than $\log P + 1$. Although this is a better performance than that of versions 1.0 and 2.0, nevertheless it does not seem fruitful to attempt to speed-up a single key search algorithm. The sequential binary search is quite fast already having logarithmic complexity.

Experiments on the Neptune system (see Tables 4.16 and 4.17) show that this version also has, on average, a poor performance. As expected, while the static shared data overheads are of the same order of magnitude as those of versions 1.0 and 2.0, the static parallel control overhead is extremely high (around 40%).

TABLE 4.16: THE EXPERIMENTAL RESULTS OF THE PARALLEL
BINARY SEARCH ALGORITHM VERSION 3.0

IKEY	T1	T2	T3	T5	SP2	SP3	SP4	IKEY	T1	T2	T3	T5	SP2	SP3	SP4
107	60.500	105.400	81.800	68.200	.574	.740	.687	4205	64.000	94.200	85.800	88.000	.679	.746	.727
122	59.100	107.500	82.500	65.600	.550	.708	.682	4223	66.000	110.600	85.300	88.900	.597	.774	.742
376	50.200	93.800	85.600	98.200	.535	.586	.569	4233	61.400	92.600	84.500	69.700	.663	.727	.981
463	63.500	94.600	83.200	68.300	.671	.763	.930	4280	51.500	93.300	85.700	71.200	.552	.601	.723
534	58.300	104.300	82.800	59.700	.559	.696	.834	4409	63.900	110.100	86.100	88.500	.580	.742	.722
610	58.200	93.200	67.400	86.900	.624	.864	.670	4558	63.800	109.700	85.900	70.100	.582	.743	.910
675	61.000	99.100	84.500	69.400	.685	.722	.879	4579	65.000	96.100	85.400	89.100	.676	.761	.730
754	60.900	90.500	84.600	68.900	.673	.720	.884	4605	67.900	94.900	87.000	90.200	.715	.780	.753
783	62.200	91.200	84.600	68.300	.682	.733	.911	4741	62.700	90.500	85.700	70.700	.693	.732	.887
799	63.200	59.100	83.700	67.500	1.069	.755	.936	4776	51.200	72.800	87.000	90.100	.703	.589	.568
1097	59.800	90.400	50.500	88.000	.662	1.184	.680	4900	35.500	91.100	86.300	70.800	.390	.411	.501
1108	53.500	106.600	85.800	48.600	.502	.624	.780	4802	61.100	70.900	86.100	71.300	.672	.710	.857
1170	58.400	88.000	84.300	69.800	.664	.693	.837	4845	66.500	93.700	86.900	70.200	.710	.765	.947
1310	60.800	59.300	84.700	86.900	1.025	.718	.700	4882	60.900	89.700	86.400	70.900	.679	.705	.859
1412	53.600	108.200	84.700	68.900	.495	.633	.778	4903	65.000	72.700	86.200	88.900	.894	.754	.731
1419	62.400	91.700	84.300	84.500	.680	.740	.721	4947	65.100	104.900	85.800	70.900	.621	.759	.918
1466	62.100	107.000	84.300	69.600	.580	.737	.892	5162	60.900	91.000	85.800	87.500	.669	.710	.696
1490	60.700	91.300	84.300	68.800	.665	.720	.882	5183	65.900	92.400	86.300	68.800	.713	.764	.958
1512	51.300	92.300	84.000	69.300	.556	.611	.745	5206	62.300	93.700	87.400	70.900	.665	.713	.879
1523	64.800	92.400	84.800	87.200	.701	.764	.712	5289	63.600	87.400	69.500	71.100	.728	.915	.895
1611	62.200	90.800	66.900	97.300	.685	.930	.701	5386	60.900	91.200	85.400	88.300	.668	.713	.690
1633	60.600	93.800	84.100	86.500	.646	.721	.701	5394	60.800	90.500	86.600	69.600	.672	.702	.874
1655	64.800	97.100	85.200	79.900	.667	.761	.914	5485	66.400	109.100	82.800	89.800	.609	.802	.739
1793	59.100	95.100	85.800	66.800	.621	.689	.885	5525	65.100	77.300	84.600	71.600	.842	.770	.909
1829	62.100	109.000	84.200	69.300	.570	.738	.896	5553	64.600	91.700	67.800	70.900	.704	.953	.911
1958	62.100	109.500	85.700	69.700	.567	.725	.891	5726	64.800	61.900	50.700	70.000	1.047	1.278	.926
1980	58.600	94.500	85.300	89.100	.620	.687	.665	5923	65.200	92.300	85.400	70.500	.706	.763	.925
2079	62.200	110.500	67.400	86.600	.563	.923	.718	6292	56.000	92.600	86.100	70.900	.605	.650	.790
2119	61.100	74.900	85.700	68.800	.816	.713	.888	6371	65.100	94.600	83.400	70.300	.688	.781	.926
2201	61.000	94.600	84.000	69.900	.645	.724	.873	6439	64.800	76.400	68.600	70.500	.848	.945	.919
2234	60.900	107.400	85.900	88.300	.567	.709	.696	6744	52.400	94.800	87.800	71.400	.553	.597	.734
2363	63.600	92.900	82.400	87.600	.685	.772	.726	6798	63.600	109.100	86.700	52.400	.583	.734	1.214
2454	60.800	74.400	84.800	69.600	.817	.717	.874	6888	53.700	79.400	85.400	72.700	.676	.629	.739
2536	51.300	95.000	84.500	88.200	.540	.607	.582	7034	66.200	110.200	84.000	72.600	.601	.788	.912
2563	60.000	91.500	85.500	68.800	.656	.702	.872	7161	68.900	98.700	85.900	73.000	.698	.802	.944
2672	48.900	91.100	86.000	48.900	.537	.569	.710	7240	51.100	93.600	86.300	88.400	.546	.592	.578
2933	64.700	106.900	84.900	87.500	.605	.762	.739	7341	66.500	90.700	87.200	89.500	.733	.763	.743
2998	63.200	108.400	84.800	87.900	.583	.745	.719	7345	64.700	90.800	86.100	70.500	.713	.751	.918
3021	64.700	91.500	84.900	87.700	.707	.762	.738	7375	67.300	91.600	85.500	89.900	.735	.787	.749
3089	59.700	91.700	83.700	68.100	.651	.713	.877	7388	59.900	91.000	85.800	88.600	.658	.698	.676
3092	53.500	92.400	85.400	69.200	.579	.626	.773	7512	53.700	111.400	84.300	89.000	.482	.637	.603
3096	48.900	95.200	85.200	69.000	.514	.574	.709	7804	61.200	96.400	86.200	89.600	.635	.710	.683
3141	61.100	107.900	86.000	69.400	.566	.710	.880	7855	68.700	108.000	67.300	89.700	.636	1.021	.766
3148	55.000	89.900	84.100	87.800	.612	.654	.626	7895	65.800	91.700	84.200	70.400	.718	.781	.935
7151	63.500	91.700	83.800	68.300	.692	.758	.935	8029	61.100	94.600	86.600	89.800	.646	.706	.680
3250	60.900	97.800	85.200	59.400	.694	.715	.875	9155	70.200	112.800	85.300	90.000	.622	.823	.780
3259	60.800	88.800	85.000	70.000	.686	.715	.865	AVERAGE TIMING AND SPEED-UP OVER ALL THE KEYS.							
3282	58.700	92.300	84.500	69.700	.636	.695	.605								
3324	58.700	92.300	84.500	69.700	.636	.695	.605	AVE.T1	AVE.T2	AVE.T3	AVE.T4	A.SP2	A.SP3	A.SP4	
3364	54.700	90.200	80.200	49.400	.606	.642	.758	60.666	94.021	83.274	77.072	.655	.716	.798	
3785	63.700	107.500	84.700	87.500	.589	.747	.723								
7400	50.100	89.100	83.200	87.100	.569	.587	.577								
7818	64.100	93.500	81.000	71.100	.707	.749	.935								
4101	61.700	84.300	84.600	49.900	.650	.725	.877								
4174	59.800	91.400	85.700	49.700	.654	.701	.958								

TABLE 4.17: THE MEASURED STATIC OVERHEADS OF THE PARALLEL
BINARY SEARCH ALGORITHM VERSION 3.0

IKEY	T1	TS	TN	SDO	PCO	IKEY	T1	TS	TN	SDO	PCO
107	60.600	33.500	33.900	.64	44.06	4194	59.600	32.600	32.900	.50	44.80
122	59.300	32.200	32.600	.67	45.03	4205	63.700	36.800	37.100	.47	41.76
376	50.300	27.700	28.000	.60	44.33	4223	65.800	38.800	39.100	.46	40.58
463	63.500	36.500	36.900	.63	41.89	4233	61.200	34.100	34.600	.82	43.46
534	58.400	31.400	31.600	.34	45.89	4290	51.300	28.800	29.000	.39	43.47
410	58.300	31.300	31.500	.34	45.97	4409	63.600	36.600	37.000	.63	41.82
473	61.100	34.000	34.400	.65	43.70	4558	63.600	36.600	36.900	.47	41.98
754	61.000	34.000	34.300	.49	43.77	4579	64.800	37.900	38.200	.46	41.05
783	62.400	35.300	35.600	.48	42.95	4605	67.600	40.600	41.000	.59	39.35
799	63.300	36.200	36.500	.47	42.34	4741	62.300	35.300	35.700	.64	42.70
1097	59.900	32.700	33.100	.67	44.74	4776	51.000	28.600	28.800	.39	43.53
1108	53.700	28.800	29.100	.56	45.81	4800	35.500	19.700	19.900	.56	43.94
1170	58.400	31.400	31.600	.34	45.89	4802	60.900	33.900	34.200	.49	43.84
1310	60.800	33.700	34.100	.64	43.91	4845	66.200	39.200	39.600	.60	40.18
1412	53.600	28.800	29.100	.56	45.71	4882	60.700	33.800	34.000	.33	43.99
1419	62.300	35.300	35.600	.48	42.86	4903	64.800	37.700	38.100	.62	41.20
1466	62.100	35.000	35.300	.48	43.16	4947	65.000	38.000	38.300	.46	41.08
1490	60.700	33.600	33.900	.49	44.15	5162	60.900	33.900	34.200	.49	43.84
1512	51.300	28.700	29.000	.58	43.47	5183	65.800	38.800	39.100	.46	40.58
1523	64.700	37.600	37.900	.46	41.42	5206	62.200	35.200	35.500	.48	42.93
1611	62.100	35.100	35.400	.48	43.00	5289	63.500	36.500	36.800	.47	42.05
1633	60.600	33.600	33.800	.33	44.22	5386	60.900	33.900	34.200	.49	43.84
1655	64.800	37.700	38.100	.62	41.20	5394	60.600	33.700	34.000	.50	43.89
1793	59.000	31.900	32.200	.51	45.42	5485	66.200	39.200	39.600	.60	40.18
1829	62.000	35.000	35.400	.65	42.90	5525	64.900	38.000	38.200	.31	41.14
1959	61.900	35.000	35.200	.32	43.13	5553	64.500	37.500	37.800	.47	41.40
1980	58.500	33.800	34.100	.51	41.71	5726	64.700	37.700	38.000	.46	41.27
2079	62.100	35.100	35.300	.32	43.16	5923	64.900	38.000	38.300	.46	40.99
2119	61.100	34.000	34.300	.49	43.86	6292	56.000	31.200	31.500	.54	43.75
2201	61.000	33.900	34.200	.49	43.93	6371	64.900	38.000	38.300	.46	40.99
2234	60.900	33.800	34.200	.64	43.84	6439	64.600	37.700	38.000	.46	41.18
2361	63.400	36.400	36.800	.63	41.96	6744	52.300	29.800	30.100	.51	42.45
2454	60.600	33.600	33.900	.50	44.06	6798	63.400	36.400	36.800	.63	41.96
2536	51.200	28.600	28.900	.59	43.55	6888	53.500	31.100	31.300	.37	41.50
2543	59.900	30.900	31.200	.50	44.57	7034	66.000	39.000	39.300	.45	40.45
2632	48.800	26.700	26.900	.41	45.70	7161	68.700	41.700	42.100	.58	38.72
2673	64.600	37.600	37.900	.46	41.33	7240	51.000	28.600	28.800	.39	43.53
2689	61.200	36.100	36.400	.47	42.41	7341	66.300	39.200	39.600	.60	40.27
3021	64.700	37.600	38.000	.62	41.37	7345	64.500	37.600	37.900	.47	41.24
3088	59.600	32.600	32.900	.50	44.80	7375	67.300	40.200	40.600	.59	39.67
3602	53.400	28.700	29.000	.37	45.88	7388	59.800	35.000	35.400	.67	40.90
3606	48.800	26.200	26.500	.61	45.70	7512	53.500	31.100	31.300	.37	41.60
7141	61.000	33.900	34.200	.66	43.77	7804	61.100	36.300	36.700	.65	39.93
3148	54.800	29.100	30.300	.36	44.71	7855	68.500	41.500	41.900	.58	38.83
3151	63.300	36.300	36.700	.63	42.02	7985	65.700	36.700	37.100	.61	40.49
3250	60.800	33.700	34.100	.66	43.91	8028	61.000	36.300	36.600	.49	40.00
3282	60.700	33.700	34.000	.49	43.99	8155	70.100	43.100	43.500	.57	37.95
3324	58.500	33.200	34.100	.51	41.71						
3364	54.500	29.800	30.100	.55	44.77						
3785	63.200	36.100	36.400	.47	42.41						
3400	49.800	27.400	27.600	.40	44.58						
7838	65.800	38.800	39.300	.76	40.27						
4121	61.100	34.100	34.400	.49	43.70						
						AVERAGE OF THE SDO AND PCO.					
						AVER.SDO		AVER.PCO			
						.52		42.71			

4.4 PARALLEL JUMP SEARCH ALGORITHMS

In this section, we shall present and analyse several MIMD implementations of the classical jump searching algorithm. The jump searching or 'block searching' algorithm was first suggested by Martin in 1977, who identified many situations where the binary search algorithm is not suitable [Martin 1977]. Following the work done by Martin, Schneiderman provided an extensive description of such a method and many of its variations [Schneiderman 1978].

Jump searching algorithms, as their name suggests, jump over portions of the ordered file until the search is localised to a small block of the file. Then smaller jumps may be applied, or a sequential search algorithm is performed, until the target key is found or its absence demonstrated. For example, in a set of 100 records sorted in ascending order, the 10th, 20th, 30th ... records may be examined until the target key is reached or exceeded. In the case that target key is bypassed, a sequential search is performed on the elements of the block just passed.

Generally speaking, when the binary search can be used it is virtually impossible to find another method that can do better, however, there are several instances where the binary search algorithm is not suitable. Therefore, for these cases which include, for example, files with compressed records*, blocked records for tape oriented searches ... etc, jump searching is preferred. Some other situations where the use of jump searching is also appreciated are found in sorted files which are linked to maintain a sequential ordering.

* Producing records of variable lengths which prevent the use of the binary search

Scheiderman identified, in all, 5 different jump searching algorithms depending on the jump size value and the number of levels involved. Three of these variations are reviewed below.

The simple jump searching method requires jumps of the size of the square root of N , where N is the total number of records in the file. These jumps are optimum in the sense that the average expected cost of the search over N elements is optimum. In order to prove that \sqrt{N} is optimum let us first compute the average cost of the search $C(N)$ which is the sum of two average costs: the average number of jumps $\frac{1}{2} N/n$ and the average number of key comparisons when the sequential search is performed on a block of size $(n-1)$. Thus

$$C(N) = \frac{N}{2n} + \frac{n}{2}$$

where n represents the size of jumps. If we assume that $C(N)$ is a continuous function, then we take the first derivative function with respect to n . This yields:

$$C'(N) = -\frac{1}{2} N/n^2 + \frac{1}{2}$$

Setting the above equation to zero and solving it we get

$$n = \sqrt{N}$$

Therefore, on average, the simple jump search algorithm performs

$$\begin{aligned} C(N) &= \frac{N}{2\sqrt{N}} + \frac{\sqrt{N}}{2} \\ &= \sqrt{N} \text{ key comparisons.} \end{aligned}$$

If jump searching is applied within a block then we get the two level simple jump searching algorithm where the square root jump size is reapplied to the $(n-1)$ records in a block. Compared with the previous method, the two level simple jump search is faster by a factor almost equal to 2. The first and second jumps are of size \sqrt{N} and $N^{1/4}$ respectively. The average expected cost of such a method is

$$\begin{aligned} C(N) &= \frac{1}{2} N/\sqrt{N} + \frac{1}{2} \sqrt{N} / N^{1/4} + \frac{1}{2} N^{1/4} \\ &= \frac{1}{2} \sqrt{N} + N^{1/4} \end{aligned}$$

comparisons, which is almost half that of the simple jump search method.

If two levels of jump searching are allowed then it is no longer optimal to have the first jump as small as \sqrt{N} . Let n_1 and n_2 be the jump size of the first and the second levels respectively. The optimum jump size values when used lead to the two-level fixed jump searching algorithm which, on average, has the following search cost:

$$C(N) = \frac{1}{2} N/n_1 + \frac{1}{2} n_1/n_2 + \frac{1}{2} n_2$$

Assuming that the above function is continuous, we take the partial derivative with respect to n_1 and n_2

$$\frac{\partial C(N)}{\partial n_1} = -\frac{1}{2} N/n_1^2 + \frac{1}{2n_2}$$

$$\frac{\partial C(N)}{\partial n_2} = -\frac{1}{2} n_1/n_2^2 + \frac{1}{2}$$

By setting the two above equations to zero and solving them, we obtain

$$n_1 = N^{2/3}$$

and

$$n_2 = N^{1/3}$$

Therefore, the average search cost of the two level fixed jump search is

$$C(N) = \frac{3}{2} N^{1/3} \text{ key comparisons}$$

Consequently, this method is faster than the two previous methods.

4.4.1 IMPLEMENTATION AND ANALYSIS

Before presenting a suitable method for implementing the parallel jump search algorithms, let us examine once more the classical jump search in order to identify the main characteristics of such an approach. As it is seen, the jump search can be viewed as a sequence of 2 or 3 sub-searching problems. For instance, if a file is considered as a sequence of blocks and a block as a sequence of sub-blocks then we have the following search problems for the jump search algorithm:

1. The target block, that is the block which is most likely to contain the sought key, is searched.
2. For the two level jump search, the target sub-block is searched within the block just found during the previous phase.
3. The target sub-block (or block in the case of a single jump search) is searched for the key.

So far, we have to decide whether to speed-up the 'overall' jump search method or each one of the above 3 sub-searching problems. the former which mainly consists of splitting the original set equally amongst the P processors, each of which is applying the sequential jump search algorithm, requires the use of broadcasting in order to avoid unnecessary searches once the target key has been located. As a result of this requirement, the execution time will be increased by all the accesses and conflicts over the shared data item used for broadcasting. The latter method which consists of parallelising every sub-searching problem uses a similar technique as the parallel sequential search version 2.0 except that broadcasting is no longer required since the search is stopped as soon as the key is either found or its value exceeded. In order to keep the overheads as low as possible, we selected to parallelise every stage of the jump search method.

Recognising the fact that every single sub-searching problem of the parallel jump search algorithm is equivalent to the parallel sequential search algorithm version 2.0, the complexity analysis is then derived from the already established average speed-up (defined in 4.8) by simply altering the set size N by its appropriate value. For instance, the parallel simple jump search algorithm, which involves 2 searches, the first one locates the target block among the \sqrt{N} possible blocks and the second finds the target key among the $(\sqrt{N}-1)$ possible keys within a block, has the following average speed-ups:

$$S_p(1) \leq P \left[1 - \frac{P-1}{2\sqrt{N}} \left(1 + \ln \frac{\sqrt{N}}{P} \right) \right]$$

and

$$S_p(2) \leq P \left[1 - \frac{P-1}{2(\sqrt{N}-1)} \left(1 + \ln \frac{(\sqrt{N}-1)}{P} \right) \right]$$

respectively to phases 1 and 2 of the algorithm.

Since $(\sqrt{N}-1) < \sqrt{N}$, we obtain a bound for $S_p(1)$ as:

$$S_p(1) < P[1 - \frac{P-1}{2(\sqrt{N}-1)} (1 + \ln \frac{\sqrt{N}}{P})]$$

Similarly, noting that $\ln(\sqrt{N}-1) < \ln\sqrt{N}$ we get

$$S_p(2) < P[1 - \frac{P-1}{2(\sqrt{N}-1)} (1 + \ln \frac{\sqrt{N}}{P})]$$

Thus, the average speed-up of the parallel simple jump search algorithm is the average of the above expressions

$$S_p = \frac{S_p(1) + S_p(2)}{2}$$

$$S_p \leq P[1 - \frac{P-1}{2(\sqrt{N}-1)} (1 + \ln \frac{\sqrt{N}}{P})]$$

A similar analysis is applied for both the parallel two level simple jump search and the two level fixed jump search algorithm. In particular for the former algorithm we have computed the following speed-ups

$$S_p(1) < P[1 - \frac{P-1}{2\sqrt{N}} (1 + \ln \frac{\sqrt{N}}{P})]$$

$$S_p(2) < P[1 - \frac{P-1}{2N^{1/4}} (1 + \ln \frac{N^{1/4}}{P})]$$

and

$$S_p(3) < P[1 - \frac{P-1}{2(N^{1/4}-1)} (1 + \ln \frac{(N^{1/4}-1)}{P})]$$

respectively for phases 1, 2 and 3. The average speed-up of the algorithm is expected to be

$$S_p = \frac{S_p(1) + S_p(2) + S_p(3)}{3}$$

For the parallel two level fixed jump search, it is found that phases 1, 2 and 3 exhibit the following speed-up respectively

$$S_p(1) < P[1 - \frac{P-1}{2N^{1/3}} (1 + \ln \frac{N^{1/3}}{P})]$$

$$S_p(2) < P[1 - \frac{P-1}{2N^{1/3}} (1 + \ln \frac{N^{1/3}}{P})]$$

and

$$S_p(3) < P[1 - \frac{P-1}{2(N^{1/3}-1)} (1 + \ln \frac{N^{1/3}-1}{P})]$$

since there are $N^{1/3}$ blocks in the original set and each block contains $N^{1/3}$ sub-blocks of $N^{1/3}$ elements each.

Since $(N^{1/3}-1) < N^{1/3}$ and $\ln(N^{1/3}-1) < \ln N^{1/3}$ we have $S_p(i)$, $i = 1, 2, 3$ bounded by

$$S_p(i) \leq P[1 - \frac{P-1}{2(N^{1/3}-1)} (1 + \ln \frac{N^{1/3}}{P})]$$

which defines the average overall speed-up of the parallel two level fixed jump search algorithm.

4.4.2 EXPERIMENTAL RESULTS

The parallelisation of the sub-searching problems which are part of the jump search method requires the use of the \$DOALL-\$SPAREND parallel programming constructs for each phase of the algorithm. Each processor would jump over blocks of Pn_1 items, where n_1 is the jump size of the sequential jump method. Thus, at the end of each phase, the key is either found or bypassed. By forcing all the

processors to synchronize before performing the second (or third phase) it is possible to use a single processor to determine the target block (or sub-block) amongst the P candidates. This means that on three occasions, we had to fork and join P processes so as to allow a sequential path to execute in between two consecutive sessions. Consequently such an approach would have a high parallel control overhead since the cost of creation/termination of paths is high on the Balance 8000. Therefore, we had to find an alternative method that allows the processors to determine the target block in parallel without having to synchronize.

Since the search for the target block is between P blocks, involving at most P key comparisons, we allowed every processor to perform the same sequential path. Note that the processors are performing redundant work since a single processor could do this. However, this avoids the burden of terminating and then creating parallel paths. Figure 4.4 illustrates the search for the target block when using two processors. If blocks are, for convenience, numbered then in our example processor 1 works on odd blocks while processor 2 works on even blocks. Now assume that the key is bypassed by the processors. Then there are two possible blocks of size n_1 where the key is likely to be. To determine which one of these blocks is the target block, every processor has to back-up by n_1 records. Therefore, at most, this process has a time complexity of P which is far lower than the cost of the creation/termination of parallel paths.

For our experiments of the parallel jump searching methods we selected the Balance 8000 system since the Neptune was unavailable at that time. We also decided to use integer numbers for the ordered set to be searched since the nature of the records is irrelevant to

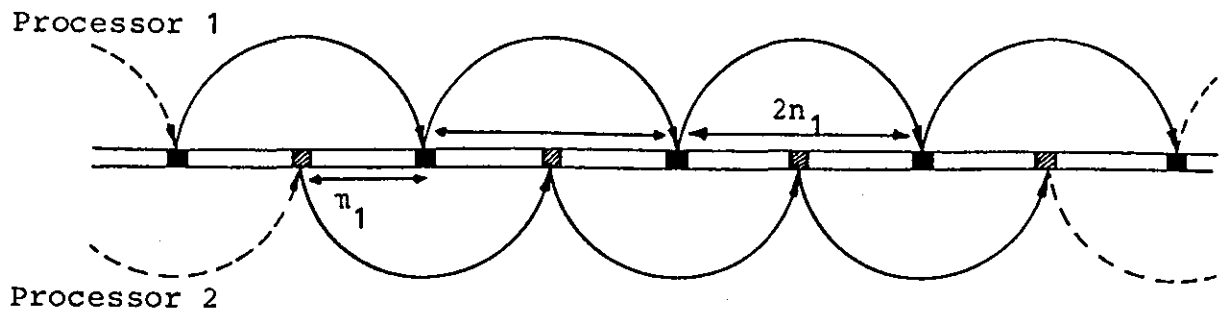


FIGURE 4.4 PARALLEL BLOCK SEARCHING WHEN $P=2$

the search method. An experiment is a series of runs of the considered parallel algorithm on 1, 2 ..., 5 processors when searching for the existence of a given key. In order to get a good representative average for the performance of a particular algorithm, we had to repeat the same run 100 times but with a different key. The 100 key locations are sampled using a uniform distribution between 1 and N . Once all the experiments relative to a specific method are complete and their timing measurements stored, we compute the average times and speed-ups which are reported in Tables 4.18, 4.19 and 4.20.

As expected, the sequential average performance of the two level fixed jump search method is better than that of the other two, while the simple jump is the worst of the three. For the parallel implementation we predicted higher speed-ups for the simple jump search and smaller speed-up for the two-level simple jump (see Figure 4.5). Experimentally this pattern is not achieved and we see that of the three the two level fixed jump search has the worst

TABLE 4.18: EXPERIMENTAL TIMING RESULTS OF THE PARALLEL SIMPLE JUMP SEARCH ALGORITHM (TIMES ARE IN MILLISECONDS)

T1	T2	T3	T4	T5	SP2	SP3	SP4	SP5	T1	T2	T3	T4	T5	SP2	SP3	SP4	SP5
5.14	2.68	1.90	1.49	1.25	1.92	2.71	3.45	4.11	6.61	3.37	2.32	1.85	1.55	1.94	2.85	3.57	4.26
5.27	2.72	1.92	1.50	1.29	1.94	2.74	3.51	4.09	5.16	2.67	1.87	1.49	1.27	1.93	2.76	3.46	4.06
2.69	1.45	1.07	.88	.74	1.86	2.51	3.06	3.64	3.54	1.90	1.35	1.03	.91	1.86	2.62	3.44	3.89
5.86	2.99	2.06	1.66	1.39	1.96	2.84	3.53	4.22	1.09	.66	.53	.47	.45	1.65	2.06	2.32	2.42
3.32	1.77	1.23	.99	.85	1.88	2.70	3.35	3.91	5.83	3.06	2.09	1.65	1.41	1.91	2.79	3.53	4.13
5.43	2.85	1.97	1.57	1.29	1.91	2.76	3.46	4.21	4.43	2.33	1.62	1.27	1.09	1.90	2.73	3.49	4.06
4.10	2.12	1.51	1.20	1.04	1.93	2.72	3.42	3.94	3.52	1.86	1.35	1.10	.89	1.89	2.61	3.20	3.96
5.02	2.64	1.82	1.41	1.22	1.90	2.76	3.56	4.11	8.15	4.25	2.85	2.30	1.84	1.92	2.86	3.54	4.43
4.69	2.42	1.75	1.36	1.17	1.94	2.68	3.45	4.01	4.22	2.23	1.58	1.27	1.09	1.89	2.67	3.32	3.87
7.05	3.60	2.49	2.03	1.76	1.96	2.83	3.47	4.01	2.02	1.13	.79	.67	.63	1.79	2.56	3.01	3.21
1.39	.77	.62	.55	.52	1.81	2.24	2.53	2.67	5.23	2.73	1.90	1.50	1.31	1.92	2.75	3.49	3.99
5.89	3.03	2.13	1.69	1.45	1.94	2.77	3.49	4.06	4.92	2.57	1.83	1.46	1.30	1.91	2.69	3.37	3.78
6.07	3.10	2.13	1.73	1.46	1.96	2.85	3.51	4.16	4.71	2.50	1.80	1.39	1.22	1.88	2.62	3.39	3.86
6.09	3.12	2.17	1.73	1.46	1.95	2.81	3.52	4.17	6.30	3.16	2.27	1.80	1.50	1.99	2.78	3.50	4.20
4.43	2.36	1.67	1.30	1.10	1.88	2.65	3.41	4.03	5.69	2.95	2.08	1.64	1.44	1.93	2.74	3.47	3.95
4.90	2.53	1.79	1.39	1.16	1.94	2.74	3.53	4.22	4.01	2.54	1.83	1.39	1.15	1.89	2.63	3.46	4.18
3.61	1.90	1.36	1.08	1.01	1.90	2.65	3.34	3.57	5.26	2.72	1.90	1.46	1.23	1.93	2.77	3.60	4.28
4.42	2.30	1.64	1.35	1.18	1.92	2.70	3.27	3.75	5.71	2.94	2.03	1.64	1.33	1.94	2.81	3.48	4.29
2.79	1.50	1.12	.93	.80	1.86	2.49	3.00	3.49	5.66	2.89	2.00	1.60	1.36	1.96	2.83	3.54	4.16
3.79	1.98	1.41	1.15	.95	1.91	2.69	3.30	3.99	5.49	2.82	1.94	1.56	1.33	1.95	2.83	3.52	4.13
6.59	3.40	2.32	1.81	1.62	1.94	2.84	3.64	4.07	3.39	1.79	1.29	1.04	.91	1.89	2.63	3.26	3.73
7.35	3.68	2.56	2.00	1.77	2.00	2.87	3.68	4.15	2.73	1.47	1.07	.88	.79	1.86	2.55	3.10	3.46
5.91	3.11	2.16	1.76	1.46	1.90	2.74	3.36	4.05	3.55	1.88	1.34	1.10	.94	1.89	2.65	3.23	3.78
3.03	1.59	1.12	.92	.86	1.91	2.71	3.29	3.52	4.69	2.49	1.81	1.40	1.26	1.88	2.59	3.35	3.72
4.58	2.42	1.67	1.36	1.16	1.89	2.74	3.37	3.93	5.11	2.72	1.87	1.52	1.30	1.88	2.73	3.36	3.93
4.84	2.59	1.79	1.38	1.21	1.87	2.70	3.51	4.00	4.89	2.53	1.80	1.45	1.22	1.93	2.72	3.37	4.01
5.45	2.81	1.92	1.55	1.33	1.94	2.84	3.52	4.10	3.75	2.06	1.40	1.15	1.02	1.82	2.68	3.26	3.68
1.95	1.09	.81	.69	.62	1.79	2.41	2.83	3.15	2.79	1.49	1.09	.90	.82	1.87	2.56	3.10	3.40
3.44	1.83	1.31	1.07	1.00	1.88	2.63	3.21	3.44	4.24	2.21	1.57	1.26	1.09	1.92	2.70	3.37	3.89
2.71	1.41	1.03	.92	.81	1.92	2.63	2.95	3.35	4.08	2.14	1.52	1.22	1.04	1.91	2.68	3.34	3.92
7.36	6.43	4.44	2.01	2.42	1.14	1.66	3.66	3.04	3.45	1.92	1.34	1.12	.98	1.90	2.72	3.26	3.72
3.48	1.85	1.33	1.13	.90	1.88	2.62	3.08	3.87	5.77	2.95	2.11	1.72	1.41	1.96	2.73	3.35	4.09
4.30	2.27	1.57	1.30	1.05	1.89	2.74	3.31	4.10	4.31	2.25	1.56	1.24	1.08	1.92	2.76	3.48	3.99
2.89	1.55	1.09	.95	.80	1.86	2.65	3.04	3.61	6.37	5.31	2.71	1.72	1.46	1.20	2.35	3.70	4.36
4.39	2.32	1.61	1.30	1.14	1.89	2.73	3.38	3.85	4.69	2.47	1.79	1.38	1.22	1.90	2.62	3.40	3.84
6.85	3.44	4.11	1.90	1.60	1.99	1.67	3.61	4.28	6.35	3.36	4.38	1.77	1.51	1.89	1.45	3.59	4.21
3.47	1.84	1.31	1.07	.92	1.89	2.65	3.24	3.77	5.00	3.56	2.49	1.44	1.26	1.40	2.01	3.47	3.97
4.13	2.19	1.56	1.30	1.07	1.89	2.65	3.18	3.86	5.64	2.90	2.03	1.64	1.37	1.94	2.78	3.44	4.12
3.54	1.88	1.33	1.05	.94	1.88	2.66	3.37	3.77	5.67	3.16	3.98	1.57	1.48	1.79	1.42	3.61	3.83
6.55	4.21	2.41	1.79	1.52	1.56	2.72	3.66	4.31	5.80	3.07	2.11	1.71	1.43	1.89	2.75	3.39	4.06
3.16	1.70	1.25	1.00	.89	1.86	2.53	3.16	3.55	3.59	1.90	1.35	1.09	.93	1.89	2.66	3.29	3.86
4.05	2.19	1.49	1.26	1.07	1.85	2.72	3.21	3.79	6.16	3.18	2.21	1.77	1.47	1.94	2.79	3.48	4.19
3.43	1.91	1.28	1.04	.91	1.80	2.68	3.30	3.77	3.55	1.90	1.36	1.04	.92	1.87	2.61	3.41	3.86
3.98	2.07	1.48	1.24	1.01	1.92	2.69	3.21	3.94	1.85	1.04	.71	.65	.60	1.78	2.61	2.85	3.08
3.02	1.66	1.18	.97	.86	1.82	2.56	3.11	3.51	4.13	2.18	1.46	1.21	1.05	1.89	2.83	3.41	3.93
8.11	4.12	3.75	2.16	1.87	1.97	2.16	3.75	4.34	3.64	1.90	1.33	1.12	.95	1.92	2.74	3.25	3.83
4.93	3.80	1.81	1.48	1.31	1.30	2.72	3.33	3.76	1.08	.65	.52	.47	.44	1.66	2.08	2.30	2.45
3.25	1.73	1.24	1.01	.89	1.88	2.62	3.22	3.65	5.82	3.00	2.11	1.67	1.42	1.94	2.76	3.49	4.10
7.77	5.96	2.75	2.04	1.69	1.30	2.83	3.81	4.60	AVERAGE								
7.45	3.86	2.63	2.07	1.73	1.93	2.83	3.60	4.31									
4.83	2.53	1.79	1.43	1.16	1.91	2.70	3.38	4.16									
3.76	1.98	1.44	1.10	.97	1.90	2.61	3.42	3.88	4.43	2.52	1.80	1.36	1.17	1.876	2.62	3.35	3.88

TABLE 4.19: EXPERIMENTAL TIMING RESULTS OF THE PARALLEL
TWO LEVEL SIMPLE JUMP SEARCH ALGORITHM
(TIMES ARE IN MILLISECONDS)

T1	T2	T3	T4	T5	SP2	SP3	SP4	SP5	T1	T2	T3	T4	T5	SP2	SP3	SP4	SP5
2.24	1.31	1.02	.86	.84	1.71	2.20	2.60	2.67	3.35	1.78	1.31	1.07	1.04	1.88	2.56	3.13	3.22
4.01	2.21	1.64	1.39	1.19	1.81	2.45	2.88	3.37	3.22	1.76	1.30	1.11	1.01	1.83	2.48	2.90	3.19
3.85	2.10	1.56	1.31	1.17	1.83	2.47	2.94	3.29	.78	.58	.54	.55	.52	1.34	1.44	1.42	1.50
4.15	2.23	1.62	1.37	1.28	1.86	2.56	3.03	3.24	.73	.53	.51	.48	.55	1.33	1.43	1.52	1.33
2.36	1.38	1.02	.96	.87	1.71	2.31	2.46	2.71	2.88	1.60	1.26	1.03	.97	1.80	2.29	2.80	2.97
1.94	1.12	.89	.77	.74	1.73	2.18	2.52	2.62	1.13	.71	.61	.57	.65	1.59	1.85	1.98	1.74
5.20	2.73	2.01	1.60	1.39	1.90	2.59	3.25	3.74	2.72	1.55	1.13	.99	.93	1.75	2.41	2.75	2.92
.76	.57	.49	.48	.54	1.33	1.55	1.58	1.41	.96	.64	.62	.56	.55	1.50	1.55	1.71	1.75
1.47	.92	.69	.70	.67	1.60	2.13	2.10	2.19	5.56	3.95	2.04	1.73	1.50	1.41	2.73	3.21	3.71
2.96	1.66	1.25	1.11	.97	1.78	2.37	2.67	3.05	2.84	1.63	1.25	1.09	.99	1.74	2.27	2.61	2.87
2.55	1.51	1.11	.99	.83	1.69	2.30	2.58	3.07	2.30	1.34	1.04	.94	.82	1.72	2.21	2.45	2.80
1.77	1.06	.80	.71	.76	1.67	2.21	2.49	2.33	3.95	2.08	1.51	1.22	1.08	1.90	2.62	3.24	3.66
1.44	.83	.65	.58	.62	1.73	2.22	2.48	2.32	2.49	1.41	1.07	.94	.85	1.77	2.33	2.65	2.93
2.32	1.36	1.00	.91	.83	1.71	2.32	2.55	2.80	1.62	.97	.76	.73	.68	1.67	2.13	2.22	2.38
3.68	2.00	1.54	1.26	1.15	1.84	2.39	2.92	3.20	2.14	1.27	1.00	.90	.76	1.69	2.14	2.38	2.82
5.01	2.64	1.94	1.54	1.37	1.90	2.58	3.25	3.66	3.88	2.12	1.53	1.29	1.19	1.83	2.54	3.01	3.26
2.99	1.70	1.34	1.13	1.00	1.76	2.23	2.65	2.99	2.55	1.52	1.12	1.01	.92	1.68	2.28	2.52	2.77
3.67	2.02	1.48	1.24	1.13	1.82	2.48	2.94	3.25	2.65	1.48	1.19	.99	.89	1.79	2.23	2.68	2.98
4.28	2.32	1.73	1.46	1.25	1.84	2.47	2.93	3.42	4.23	2.30	1.66	1.43	1.25	1.84	2.55	2.96	3.38
1.68	1.04	.74	.78	.73	1.62	2.27	2.15	2.30	4.88	2.58	1.86	1.50	1.28	1.89	2.62	3.25	3.81
1.06	.65	.57	.57	.64	1.63	1.86	1.86	1.66	.60	.50	.45	.45	.53	1.20	1.33	1.33	1.13
2.33	1.32	1.03	.88	.82	1.77	2.26	2.65	2.84	4.63	2.50	1.83	1.57	1.30	1.85	2.53	2.95	3.56
3.20	1.77	1.44	1.20	1.04	1.81	2.22	2.67	3.08	4.00	2.25	1.59	1.32	1.24	1.78	2.52	3.03	3.23
3.77	2.11	1.55	1.33	1.23	1.79	2.43	2.83	3.07	2.70	1.50	1.18	1.05	.90	1.80	2.29	2.57	3.00
4.26	2.31	1.66	1.45	1.27	1.84	2.57	2.94	3.35	.68	.49	.43	.49	.51	1.39	1.58	1.39	1.33
3.93	2.16	1.69	1.36	1.22	1.82	2.33	2.89	3.22	5.41	2.80	1.96	1.69	1.39	1.93	2.76	3.20	3.89
3.55	1.95	1.48	1.20	1.14	1.82	2.40	2.96	3.11	5.41	2.86	2.12	1.69	1.47	1.89	2.55	.46	3.68
4.50	2.47	1.88	1.45	1.34	1.82	2.39	3.10	3.36	3.13	1.75	1.31	1.80	1.01	1.79	2.39	1.74	3.10
3.49	1.93	1.42	1.24	1.13	1.81	2.46	2.81	3.09	2.82	1.60	1.15	.98	1.00	1.76	2.45	2.88	2.82
4.54	2.42	1.81	1.51	1.31	1.88	2.51	3.01	3.47	.91	.60	.56	.57	.59	1.52	1.63	1.60	1.54
2.81	1.58	1.19	.99	.95	1.78	2.36	2.84	2.96	3.28	1.84	1.39	1.14	1.04	1.78	2.36	2.88	3.15
2.14	1.24	.99	.80	.81	1.73	2.16	2.68	2.64	3.42	1.93	1.36	1.22	1.13	1.77	2.51	2.80	3.03
3.46	1.89	1.41	1.22	1.10	1.83	2.45	2.84	3.15	2.48	1.35	1.14	.92	.82	1.84	2.18	2.70	3.02
2.12	1.22	.95	.78	.77	1.74	2.23	2.72	2.75	2.16	1.29	.98	.87	.77	1.67	2.20	2.48	2.81
4.15	2.25	1.59	1.38	1.26	1.84	2.61	3.01	3.29	1.11	.67	.58	.60	.59	1.66	1.91	1.85	1.88
3.39	1.88	1.41	1.19	1.01	1.80	2.40	2.85	3.36	1.32	.82	.64	.65	.61	1.61	2.06	2.03	2.16
2.67	1.52	1.15	.93	.89	1.76	2.32	2.87	3.00	2.70	1.53	1.19	.95	.98	1.76	2.27	2.84	2.76
3.31	1.83	1.43	1.16	1.09	1.81	2.31	2.85	3.04	1.67	1.04	.75	.74	.72	1.61	2.23	2.20	2.32
1.35	.78	.72	.64	.60	1.73	1.88	2.11	2.25	5.51	4.71	2.03	2.18	1.42	1.17	2.71	2.53	3.88
3.76	2.09	1.55	1.32	1.16	1.80	2.43	2.85	3.24	1.17	.71	.63	.61	.64	1.65	1.86	1.92	1.83
4.59	2.48	1.75	1.46	1.30	1.85	2.62	3.14	3.53	4.77	2.63	1.85	1.56	1.37	1.81	2.58	3.06	3.48
.94	.59	.55	.56	.55	1.59	1.71	1.68	1.71	.55	.43	.41	.42	.53	1.28	1.34	1.31	1.04
5.22	2.75	2.09	1.66	1.45	1.90	2.50	3.14	3.60	.57	.47	.49	.51	.49	1.21	1.16	1.12	1.16
.91	.59	.53	.55	.56	1.54	1.72	1.65	1.63	2.75	1.54	1.21	1.00	.99	1.79	2.27	2.75	2.78
5.43	2.85	2.06	1.66	1.50	1.91	2.64	3.27	3.62	4.57	2.31	1.69	1.32	1.17	1.98	2.70	3.46	3.91
.87	.59	.53	.56	.52	1.47	1.64	1.55	1.67	4.52	2.52	1.85	1.45	1.27	1.79	2.44	3.12	3.56
1.21	.74	.63	.61	.59	1.64	1.92	1.98	2.05	2.37	1.36	1.12	.92	.82	1.74	2.12	2.58	2.89
3.94	2.15	1.60	1.34	1.13	1.83	2.46	2.94	3.49	1.95	1.15	.96	.79	.74	1.70	2.03	2.47	2.64
2.77	1.58	1.19	1.04	.94	1.75	2.33	2.66	2.95	AVERAGE								
4.99	2.69	1.98	1.60	1.39	1.86	2.52	3.12	3.59									
4.45	2.38	1.75	1.41	1.29	1.87	2.54	3.16	3.45									
3.04	1.70	1.30	1.10	1.02	1.79	2.34	2.76	2.98	2.93	1.66	1.24	1.17	.97	1.72	2.25	2.56	2.83

TABLE 4.20: EXPERIMENTAL TIMING RESULTS OF THE PARALLEL
TWO LEVEL FIXED JUMP ALGORITHM
(TIMES ARE IN MILLISECONDS)

T1	T2	T3	T4	T5	SP2	SP3	SP4	SP5	T1	T2	T3	T4	T5	SP2	SP3	SP4	SP5
.55	.46	.42	.43	.72	1.20	1.31	1.28	.76	1.08	.71	.61	.44	.66	1.52	1.77	2.45	1.64
1.36	.78	.70	.73	.96	1.74	1.94	1.86	1.42	1.11	.70	.62	.62	.62	1.59	1.79	1.79	1.79
1.25	.73	.62	.62	1.13	1.71	2.02	2.02	1.11	1.19	.79	.61	1.68	.84	1.51	1.95	.71	1.42
1.52	.92	.73	.74	.75	1.65	2.08	2.05	2.03	.83	.55	.51	.54	.93	1.51	1.63	1.54	.89
1.01	.64	.59	.62	.87	1.58	1.71	1.63	1.14	1.49	.86	.69	.44	.64	1.73	2.16	3.39	2.33
.99	.64	.61	.57	.54	1.55	1.62	1.74	1.83	.79	.55	.51	.56	.64	1.44	1.55	1.41	1.23
1.75	1.03	.87	1.06	1.02	1.70	2.01	1.65	1.72	.77	.56	.50	.64	.62	1.37	1.54	1.20	1.24
.82	.60	.51	.46	1.07	1.37	1.61	1.78	.77	1.38	.86	.71	.61	1.03	1.60	1.94	2.26	1.34
1.17	.72	.65	.70	.74	1.62	1.80	1.67	1.58	1.71	1.78	.70	.46	.89	.96	2.44	3.72	1.92
1.20	.76	.65	.59	.73	1.58	1.85	2.03	1.64	.94	.69	.57	.66	.71	1.36	1.65	1.42	1.32
.99	.66	.56	.54	1.02	1.50	1.77	1.83	.97	1.00	.65	.57	.66	.52	1.54	1.75	1.52	1.92
.61	.48	.44	.71	.60	1.27	1.39	.86	1.02	1.08	.69	.57	.76	.59	1.57	1.89	1.42	1.83
.84	.37	.36	.44	.61	1.19	1.22	1.00	.72	1.28	.79	.72	.52	.72	1.62	1.78	2.46	1.78
1.30	.84	.68	.65	1.08	1.55	1.91	2.00	1.20	1.36	.82	.64	.56	.80	1.66	2.13	2.43	1.70
.98	.64	.57	.58	.78	1.53	1.72	1.69	1.26	1.26	.78	.65	.69	1.00	1.62	1.94	1.83	1.26
1.04	.66	.57	.66	.63	1.58	1.82	1.58	1.65	1.62	.97	.77	.72	.66	1.67	2.10	2.25	2.45
1.21	.74	.66	.68	.62	1.64	1.83	1.78	1.95	1.07	.68	.60	.45	1.00	1.57	1.78	2.38	1.07
1.21	.72	.62	.61	.73	1.68	1.95	1.98	1.66	1.27	.76	.65	.75	.78	1.67	1.95	1.69	1.63
1.57	.91	.73	.99	1.20	1.73	2.15	1.59	1.31	1.34	.79	.71	.72	.72	1.70	1.89	1.86	1.86
.78	.55	.51	.48	.78	1.42	1.53	1.63	1.00	1.92	1.12	.85	.57	.69	1.71	2.26	3.37	2.78
.94	.61	.57	.57	.71	1.54	1.65	1.65	1.32	.94	.62	.56	.55	.87	1.52	1.68	1.71	1.08
1.35	.83	.72	.76	.65	1.63	1.88	1.78	2.08	1.43	.88	.73	.60	.79	1.63	1.96	2.38	1.81
1.09	.66	.59	.66	.79	1.65	1.85	1.65	1.38	1.23	.73	.60	.69	.88	1.68	2.05	1.78	1.40
.79	.52	.54	.63	.60	1.52	1.46	1.25	1.32	1.21	.79	.64	.76	.68	1.53	1.89	1.59	1.78
1.21	.75	.60	.77	.84	1.61	2.02	1.57	1.44	.69	.49	.47	.58	1.17	1.41	1.47	1.19	.59
.90	.58	.52	.56	.69	1.55	1.73	1.61	1.30	1.34	1.73	.62	.72	.67	.77	2.16	1.86	2.00
1.27	.77	.66	.60	.75	1.65	1.92	2.12	1.69	1.08	1.36	.63	.56	1.44	.79	1.71	1.93	.75
1.01	.63	.60	.56	.85	1.60	1.68	1.80	1.19	1.27	.81	.63	.68	1.24	1.57	2.02	1.87	1.02
1.10	.70	.63	.55	.83	1.57	1.75	2.00	1.33	.74	.50	.47	.69	.60	1.48	1.57	1.07	1.23
1.57	.96	.85	.79	1.03	1.64	1.85	1.99	1.52	1.15	.72	.63	.74	.70	1.60	1.83	1.55	1.64
1.33	.81	.73	.73	.89	1.64	1.82	1.82	1.49	1.14	.73	.62	.56	.69	1.56	1.84	2.04	1.65
1.14	.71	.59	.54	.81	1.61	1.93	2.11	1.41	1.40	.81	.66	.62	1.20	1.73	2.12	2.26	1.17
1.44	.90	.70	.71	.90	1.60	2.06	2.03	1.80	1.69	1.00	.79	.61	.52	1.69	2.14	2.77	3.25
.99	.66	.58	.47	.57	1.50	1.71	2.11	1.74	.71	.53	.49	1.00	.60	1.34	1.45	.71	1.18
1.25	.75	.64	.67	.63	1.67	1.95	1.87	1.98	.55	.42	.42	.44	.96	1.31	1.31	1.25	.57
1.31	.93	.74	.69	.98	1.62	2.04	2.19	1.54	1.49	.89	.76	1.96	1.06	1.67	1.96	.76	1.41
1.65	.98	.82	.73	.72	1.68	2.01	2.26	2.29	.93	.62	.60	.42	1.05	1.50	1.55	2.21	.89
1.25	.80	.67	.76	.76	1.56	1.87	1.64	1.64	1.28	.81	.69	.62	.97	1.58	1.86	2.06	1.32
1.01	.69	.58	.84	.96	1.46	1.74	1.20	1.05	1.81	1.12	.83	.43	.52	1.62	2.18	4.21	3.48
.99	.65	.57	.70	.61	1.52	1.74	1.41	1.62	1.24	.76	.66	.63	1.06	1.63	1.88	1.97	1.17
1.47	.86	.71	.66	.74	1.71	2.07	2.23	1.99	1.56	.85	.64	.67	.99	1.84	2.44	2.33	1.58
.51	.45	.42	.57	.63	1.13	1.21	.89	.81	1.00	.61	.52	.87	.60	1.64	1.92	1.15	1.67
1.39	.85	.66	1.66	.72	1.64	2.11	.84	1.93	.87	.58	.50	.67	1.11	1.50	1.74	1.30	.58
.47	.37	.40	.43	.70	1.27	1.17	1.09	.67	1.17	.76	.67	.58	.88	1.54	1.75	2.02	1.33
1.66	1.03	.71	.57	1.08	1.61	2.34	2.91	1.54	1.20	.72	.60	1.29	.91	1.67	2.00	.93	1.32
.71	.53	.47	.43	.96	1.34	1.51	1.65	.74	1.54	.88	.74	.55	.59	1.75	2.09	2.80	2.61
1.48	.86	.73	.63	.88	1.72	2.03	2.35	1.68	1.00	.67	.57	.62	.69	1.49	1.75	1.61	1.45
1.30	.77	.66	.61	.89	1.69	1.97	2.13	1.46	1.54	.90	.73	.56	1.08	1.71	2.11	2.75	1.43
1.15	.69	.61	.68	.69	1.67	1.89	1.69	1.67	AVERAGE								
1.95	1.15	.85	.66	.68	1.70	2.29	2.95	2.87									
1.86	1.12	.87	.58	.96	1.66	2.14	3.21	1.94									
.81	.56	.53	1.43	.96	1.45	1.53	.57	.84	1.18	.76	.63	.68	.82	1.55	1.84	1.85	1.51

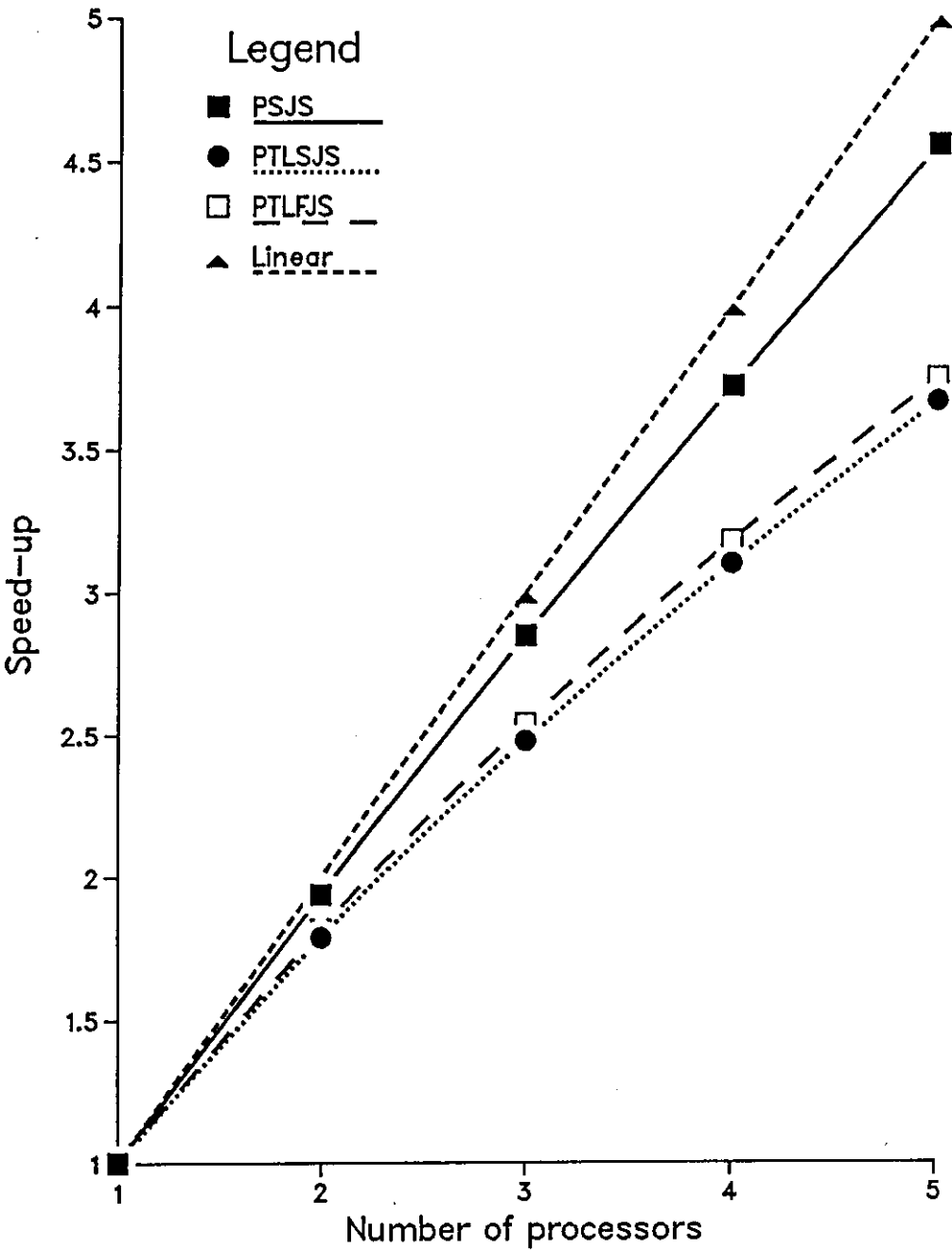


FIGURE 4.5: THEORETICAL SPEED-UPS OF THE PARALLEL JUMP SEARCH ALGORITHMS

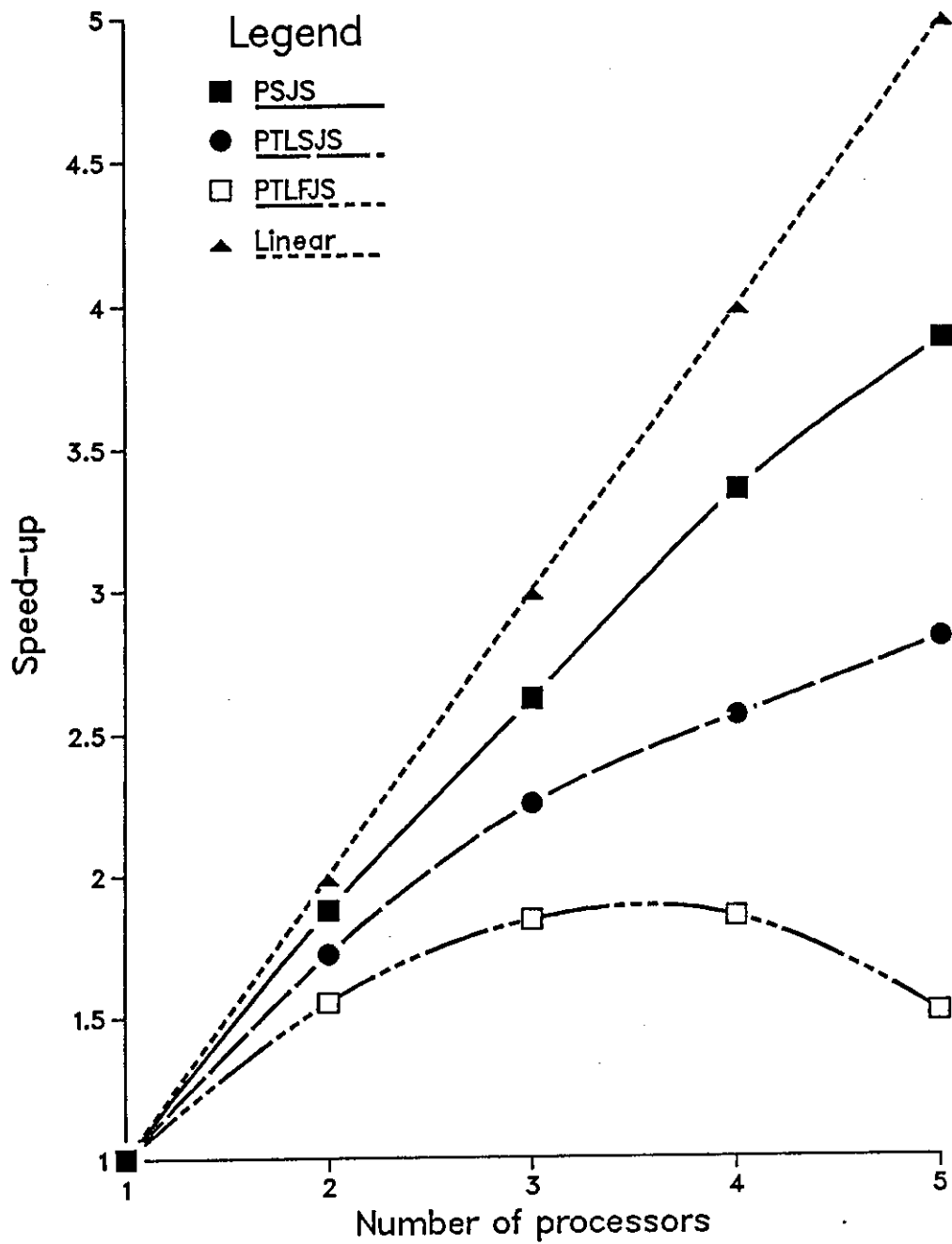


FIGURE 4.6: EXPERIMENTAL SPEED-UPS OF THE PARALLEL JUMP SEARCH ALGORITHMS

parallel performance (see Figure 4.6). However, this is not surprising since this method performs on average less operations than the other two sequentially and when using P processors the amount of work performed by each processor is even smaller. However, these methods are expected to do better with very large sets. In Tables 4.21 and 4.22 we report the maximal number of key comparisons performed in each stage by each method when varying the number of processors. A sample of five random keys is selected.

Key	Processors				
	1	2	3	4	5
1	89, 15, 4	45, 8, 2	30, 5, 2	22, 4, 1	18, 3, 1
2	181, 13, 6	91, 7, 3	61, 4, 2	45, 3, 2	37, 3, 2
3	176, 8, 7	88, 4, 4	59, 3, 3	44, 2, 2	36, 2, 2
4	191, 6, 9	96, 3, 5	64, 2, 3	48, 2, 3	36, 1, 2
5	103, 3, 8	52, 2, 4	35, 1, 3	26, 1, 2	21, 1, 2

TABLE 4.21: Number of key comparisons performed by the parallel simple jump search method

Key	Processors				
	1	2	3	4	5
1	15, 1, 6	8, 1, 3	5, 1, 2	4, 1, 2	3, 1, 2
2	29, 20, 18	15, 10, 9	10, 7, 6	8, 5, 4	6, 4, 4
3	28, 27, 4	14, 14, 2	10, 9, 2	10, 9, 2	6, 6, 1
4	30, 41, 4	15, 21, 2	10, 14, 2	10, 14, 2	6, 9, 1
5	17, 4, 32	9, 2, 16	6, 2, 11	6, 2, 11	4, 1, 7

TABLE 4.22: Number of key comparisons performed by the parallel fixed jump search method

The tables above show that the time-complexity for backing-up the blocks which were neglected during the theoretical performance analysis has a considerable degrading effect on the performance of the two level fixed jump algorithm than on the two level simple jump search since the number of key comparisons, when more than 2, 3 processors are used, is almost equal, or even less than, P .

4.5 CONCLUSION

In this chapter we have designed and analysed several searching algorithms, each of which is suitable for a specific situation. The studied algorithms are the sequential search, the binary search and the jump or 'block' search algorithms.

Using the powerful partitioning strategy as our basic MIMD implementation method for these parallel algorithms, we have devised several different versions for the sequential and binary search algorithms. In version 1.0, the elements accessed by one processor are adjacent whereas they are distant by P elements in the second version. The third version, defined for the binary search, consists of partitioning the current sub-set into P in every iteration.

Initially we established, theoretically and experimentally, that an optimum time-complexity for the parallel sequential search is obtained when M , the number of paths, is equal to P . We then studied the effect of broadcasting the current state of the search on the average parallel performance of the search problem. Both theoretical and experimental analysis showed that broadcasting which is performed through a shared data structure, improves the average speed-up of the sequential search algorithm from $\frac{P+1}{2}$ (when

broadcasting is not considered) to superlinear and linear speed-ups with versions 1.0 and 2.0 respectively.

Sequentially, the binary search algorithm has a good performance (i.e. the time-complexity is logarithmic) which is difficult to beat in most cases. However, its parallel implementation showed rather poor performance characteristics. A fact that supports the general rule which states that "good sequential algorithms are not necessarily the best parallel algorithms". Experiments on the binary search method showed that, on average, the speed-ups of the three parallel versions are less than unity. One of the reasons for such behaviour is that versions 1.0 and 2.0 do not split the total amount of work equally between the activated processors. Except for version 3.0 which involves all the processors in every iteration of the algorithm, the work is equally partitioned. However, due to the high parallel control overheads (since every iteration creates and terminates P paths, each of which performs a single key comparison) any gain achieved by using such a method soon dies out. Therefore, we came to the conclusion that the binary search, although very efficient sequentially, is not suitable for parallel implementation.

On the other hand, the jump search algorithm which is preferred in some situations where the binary search is not suitable, achieved acceptable performance. In particular, the simple jump search method showed an almost linear speed-up. However, due to the fewer number of operations to perform in parallel as the jump size gets larger, the speed-ups of the two level simple jump search and the two level fixed jump search methods flattened quickly as P increased. In order to achieve good performance, these methods should be used with extra large sets or files.

Chapter 5

PARALLEL STRING MATCHING ALGORITHMS

5.1 INTRODUCTION

The string searching problem is at the heart of many information retrieval and text editing applications where it is required to locate some or all occurrences of a user-specified pattern in a text string in a minimal amount of both processing time and memory space.

The Brute Force search algorithm is perhaps the most obvious way to search for a matching pattern in a text string by trying to start the search at every position of the text, abandoning the search as soon as an incorrect character is found. Although this approach is very simple, it can be very inefficient with some types of patterns and strings. For example, the search for a^nb in a text of the form $a^{2n}b$ requires $(n+1)^2$ comparisons [Knuth 1977]. Furthermore, backing up the input text as we go through it, can add annoying complications if the frequently involved buffering operations are considered.

Unlike the Brute Force algorithm, the Karp-Rabin and the Knuth-Morris-Pratt algorithms avoid backing up the text. Therefore these algorithms inspect every character passed once and only once. By comparison, the Brute Force algorithm inspects on average, 1.1 characters for every referenced character [Boyer 1977].

The Karp-Rabin algorithm considers the pattern as a key of m digits and d , the alphabet size, as the base. They observed that string matching is the same problem as the standard searching problem by searching all possible m -character sub-strings in the text whose hash function value is equal to the hash function value of the pattern. The highly mathematical operations involved make the Karp-Rabin algorithm of less practical use than the Brute Force

algorithm. The Knuth-Morris-Pratt algorithm uses a basic idea that the pattern is shifted right by a concise number of places whenever a mismatch is detected. A table containing all the shift values is precomputed for every pattern.

A recent algorithm that solves the string searching problem even faster than any previous method is the Boyer-Moore algorithm which unlike its predecessors, compares the pattern with the text string from the right end. Whenever a mismatch occurs the pattern is shifted right according to a precomputed table. In the case that the text character positioned against the last character in the pattern does not appear in the pattern, the pattern is immediately shifted right by a distance equal to the pattern length (m). Thus when the alphabet size is large, on average, only n/m characters of the text are inspected, where n is the text string length [Knuth 1977]. Boyer and Moore showed that the worst-case running time is proportional to $n*m$. The running time can essentially be proportional to $n+r*m$ where r is the number of occurrences of the pattern in the text [Knuth 1977].

Knuth also described a variation of the Boyer-Moore algorithm that is faster than the original algorithm by making a small modification in the algorithm that computes the shift table.

In this Chapter, the parallel implementation of the five considered string searching algorithms are presented. The experimental results showed that the Boyer-Moore and its variation outperform by far the remaining methods. In Section 5.2 we review all the string searching algorithms and place them into perspective by giving a short and brief history. In Section 5.3 we present the parallel implementation of these algorithms and their experimental results

performed on 2 MIMD parallel computer systems available at Loughborough University with comparisons of the predicted performance. Finally Section 5.4 concludes the work reported in this chapter.

5.2 HISTORY

There is a Brute Force algorithm for string searching problems which is in wide use. It checks for each possible position in the string that could possibly match the pattern, whether it does in fact match. The method which is the first idea that comes to mind keeps a pointer (i) in the text string and another (j) in the pattern. As long as they point to matching characters both pointers are incremented by 1, otherwise j is reset to 1 and i is reset to correspond to moving the pattern right by one place. The search is successful if at the end of the algorithm j is greater than m (where m is the pattern length), otherwise it is unsuccessful. While this algorithm has a worst-case running time proportional to $n*m$, the strings that arise in many applications lead to a running time almost always proportional to $n+m$.

In 1970, S.A. Cook proved for a particular type of abstract machine that an algorithm exists which solves the string matching problem in a worst case running time proportional to $n+m$. Following the work of Cook used to prove his theorem, D.E. Knuth and V.R. Pratt designed an algorithm (not intended to be at all practical) which they were able to refine and get virtually a simple and practical algorithm. However it turned out that in the meantime J.H. Morris had virtually discovered the same algorithm as a solution to an annoying problem (of not wanting to ever back up the input text) that confronted him while implementing a text editor. The basic idea behind the

algorithm discovered by Knuth, Morris and Pratt is this: while the Brute Force algorithm forgets all about the known characters that have matched just before a mismatch is detected, Knuth-Morris-Pratt observed that there is enough information to create them and advantage should be taken somehow of this information instead of backing up the text over all those known characters. The pattern is initially put above the text string so that the left-most characters of both strings are aligned. The text is scanned from left to right without back ups. In the case of a mismatch, the pattern is shifted to the right according to a precomputed table. The computation of the shift table is short (requiring m steps) but tricky: it is basically the same algorithm as the Knuth-Morris-Pratt algorithm except that it is used to match the pattern against itself.

The Knuth-Morris-Pratt algorithm is not likely to be significantly faster than the Brute Force algorithm in most applications, because few applications search highly self-repetitive patterns in highly self-repetitive texts. However the method has a major advantage from a practical point of view. This makes the method convenient for use on a large file being read in from some external device. Algorithms that require back ups require some complicated buffering operations in this situation.

The work of Knuth-Morris-Pratt was not published until 1976, and meanwhile R.S. Boyer and J.S. Moore (and independently R.W. Gosper) discovered an algorithm which is much faster in many applications. They proved, if backing up is not a problem, a significantly faster method is obtained by scanning the pattern from right to left when trying to match the pattern against the text string. The algorithm keeps 2 pointers: (i) pointing to the current character in the text string and (j) pointing to the current character in the pattern

string. Initially j points to the last character (right most character, starting from the left) of the pattern and i is set to m . If no mismatch occurs, then an occurrence of the pattern has been found. Otherwise, the Boyer-Moore algorithm chooses the largest value of two precomputed shifts, δ_1 and δ_2 , by which the pattern has to be shifted before a new matching attempt is undertaken.

The computation of the δ_1 and δ_2 shift tables is quite an involved process to explain, but it is solely based on the pattern and the alphabet character set. The whole idea behind the Boyer-Moore algorithm could be resumed in the following remarks:

1. If the current text character positioned against the last character of the pattern does not occur in the pattern, then the pattern is shifted to the right by m positions. A shift of less than m places would not lead to a match;
2. If the last character of the pattern has an occurrence δ_1 places from the right end in the pattern, then the pattern is immediately moved δ_1 places to the right. A shift of less than δ_1 would position 2 mismatching characters. The computation of the δ_1 table requires $(m+d)$ steps where d is the alphabet size;
3. The second shift table is similar to that of the Knuth-Morris-Pratt shift table except that the order of matching the pattern against itself is from right to left. The computation of the δ_2 table is therefore $O(m)$ steps.

Boyer and Moore showed that for a large alphabet, this algorithm on average, inspects only n/m characters. The worst-case running time of $(n*m)$ was proved to be $(n+rm)$, where r is the number of

occurrences of the pattern. Recently Z. Galil [Galil 1979] suggested a variation of the Boyer-Moore algorithm that improved considerably the worst-case running time (when the pattern does not exist in the string) to $O(n)$.

This story illustrates the fact that the search for a "better algorithm" is still often justified: the advent of parallel computer systems surely opens even new horizons for this problem to be exploited.

5.3 DESIGN, ANALYSIS AND IMPLEMENTATION

In this section, we present a parallel implementation of the string searching problem for an MIMD parallel computer system. We assume that the reader is familiar with the notion of multiprocessor computer systems and parallel language tools. Our experimental results were obtained on Neptune which is a 4-processor system manufactured by Texas Instruments and on Balance 8000 which is a 6-processor (extendable to 12) system manufactured by Sequent Computer Systems Inc.

The parallel algorithm design methodology used is the powerful partitioning (also known under the name of Divide-and-Conquer) method where the initial complex problem is partitioned into a certain number of smaller and less complex sub-problems. These sub-problems are then scheduled through a list of identical processors to be executed in parallel and independent of each other. Of course, the degree of independence depends on the amount of interaction between the execution of the sub-problems. The problem solution is then obtained by merging the partial solutions of the individual sub-problems once they are all completed.

In our string searching problem, the divide-and-conquer method suggests the partition of the shared string equally amongst the P available processors. The exact number of elements (n_{elem}) in every sub-string is defined so the following points are observed:

1. Each processor has exclusive access to $n/p+m-1$ characters of the sub-string it is allocated to. This would ensure that none of these characters is inspected by more than one processor. Therefore parallel performance is maximized since the P processors are inspecting the string at P independent locations. However a price has to be paid because of the shared memory access overheads. Happily these are forecasted to be of no significant importance.
2. Two consecutive sub-strings are allowed to overlap by $m-1$ characters. The left most $m-1$ characters of a sub-string are added at the end of the preceding sub-string (except for the sub-string starting with the first character of the string). Failing to do this makes the parallel search algorithm miss all the occurrences that lay between two sub-strings. An overlap of m (or more than m) characters leads to two processors finding the same occurrence (if any). The immediate implication of this is that one of the two implied processors is performing exactly the same work as the other one. This redundancy is undesirable in the design of parallel programs.

The sub-strings defined above are scheduled through a list of P active processors to be executed in parallel and independent of each other. Each processor searches all occurrences of the given pattern in the sub-string it is associated with. Every processor stores all

the locations of the found occurrences in a local array. Once all the processor have terminated searching their sub-strings the location arrays are sorted and merged into a single array. The parallel algorithm shown below is an implementation of any sequential string searching algorithm.

\$DOALL (creation of P paths)

nelem=n/p

is=me*nelem+1

ie=(me+1)*nelem-1

Search the sub-string (is, ie) for all occurrences of the pattern using a particular sequential string search method

\$PAREND (end of a path)

Each process (or path) is numbered using a local variable (me) at the initiation phase starting from 0 to P-1. After the creation of P paths, the path creator processor (parent) as well as the other remaining processors (children) try to acquire a path to execute by entering the scheduling process critical region.

The first step of a path is the computation of a lower (is) and upper (ie) bounds of the sub-string it is about to search using its identification number (me). Then the search is started from the character indexed by (is) until the character indexed by (ie) is reached.

The testing procedure used to compare the performance of the five parallel string searching algorithm is based on the average running time since it covers a large number of possible patterns. An English text of n characters is chosen for the experiments. The pattern length is varied from 6 to 15. For every pattern length considered,

10 random patterns are searched and their average timing is computed. Table 5.1 compares the sequential performance of 4 string matching algorithms against the slowest Brute Force method on the Neptune system. Table 5.2 reports the comparative sequential results of 5 string matching algorithms performed on the Balance 8000 system. The methods are:

- BF: The Brute Force algorithm
- KR: The Karp-Rabin algorithm
- KMP: The Knuth-Morris-Pratt algorithm
- BM: The Boyer-Moore algorithm
- IBM: The Improved Boyer-Moore algorithm

TABLE 5.1: Sequential performance results performed on the Neptune System. The string size was 16000 characters. Timing unit is in seconds

m	Sequential timing of				Speed-up		
	BF	KMP	BM	IBM	KMP	BM	IBM
6	1.188	.737	.282	.278	1.614	4.403	4.482
7	1.195	.745	.240	.236	1.608	5.117	5.210
8	1.196	.748	.220	.215	1.602	5.692	5.811
9	1.197	.748	.208	.204	1.602	5.952	6.080
10	1.206	.756	.196	.919	1.598	6.313	6.465
11	1.195	.749	.170	.168	1.600	7.294	7.386
13	1.194	.743	.171	.168	1.610	7.175	7.315
14	1.190	.742	.164	.162	1.607	7.610	7.741
15	1.187	.741	.157	.154	1.605	7.760	7.927

TABLE 5.2: Sequential performance results performed on the Balance 8000 system. The string size was fixed to 500000 characters. Timing unit is in seconds

m	Sequential timing of					KR	Speed-up		
	BF	KR	KMP	BM	IBM		KMP	BM	IBM
6	43.22	58.95	39.11	4.24	4.20	.73	1.10	10.81	10.91
7	41.64	58.94	37.69	3.55	3.50	.71	1.10	12.15	12.31
8	42.76	58.94	38.71	2.96	2.90	.73	1.10	14.89	15.13
9	42.35	58.94	38.33	2.63	2.59	.72	1.10	16.49	16.78
10	41.57	58.94	37.66	2.99	2.94	.71	1.10	14.45	14.64
11	42.01	58.93	37.97	2.26	2.20	.71	1.11	19.18	19.68
12	42.69	58.94	38.55	2.48	2.40	.72	1.11	18.05	18.58
13	43.22	58.94	39.18	2.16	2.13	.73	1.10	20.09	20.42
14	43.27	58.93	39.07	2.07	1.99	.73	1.11	21.86	22.64
15	41.74	58.93	37.66	2.00	1.94	.71	1.11	21.39	21.94

From both tables, it can be seen that as the pattern length increases the Boyer-Moore average running time decreases. This point could easily be proved if we assume that the alphabet of the input string is large enough to apply the Boyer-Moore remark which is: for a large alphabet, on average only n/m characters are inspected.

Let $m1$ and $m2$ be the length of two patterns, where $m2$ is greater than $m1$. The search for all occurrences of the pattern of length $m2$ is faster than the search for all occurrences of the pattern of length $m1$ by at most:

$$\frac{\frac{n}{m1}}{\frac{n}{m2}} = \frac{m2}{m1} \text{ times}$$

For example if m_1 and m_2 are chosen to be respectively equal to 6 and 15, an expected speed of the search of the pattern of length 15 over the search of the pattern of length 6 is predicted to be 2.5. The experiments on the Balance and Neptune systems showed a speed of respectively 2.12 and 1.79.

The difference in the above theoretical and experimental figures was not unexpected since a few important factors were ignored in the above model. Some of the ignored factors are the randomness of the text and the string, the number of occurrences of the pattern and its function with the pattern length since the way the random patterns are generated for every pattern length leads to fewer occurrences with longer patterns than with short patterns. Nevertheless, the theoretical result forms an upper bound which could not possibly be reached if all the above mentioned factors were taken into account. By contrast, the BF, KMP and RK algorithms are invariant to the pattern length.

5.3.1 PARALLEL BRUTE FORCE ALGORITHM

Using one processor, the BF algorithm performance is almost always proportional to $n+rm$ because of the selection of the English text. We assume that the r occurrences of the random pattern are uniformly distributed in the text string. Thus, every sub-string of length $(n/P + m - 1)$ is expected to contain r/P occurrences of the given pattern. The time T_p it takes to run the parallel Brute Force algorithm on the P -processor computer system is given by:

$$T_p = \left(\frac{n}{P} + m - 1\right) + \left(\frac{r}{P}\right)m$$

which simplifies to:

$$T_p = \frac{n}{P} + \frac{rm}{P} + m - 1$$

The expected speed-up of the parallel Brute Force algorithm when all parallel overheads are ignored is given by:

$$S_p = \frac{T_1}{T_p} = \frac{n + rm}{\frac{n + rm}{P} + m - 1}$$

which is also equal to:

$$S_p = P \left(1 - \frac{P(m-1)}{n + (r+P)m - P} \right)$$

The expected efficiency is:

$$E_p = \frac{S_p}{P} = \left(1 - \frac{P(m-1)}{n + (r+P)m - P} \right)$$

Tables 5.3(a) and 5.3(b) give the parallel experimental performance of the Brute Force algorithm performed respectively on the Neptune and Balance systems.

The Brute Force algorithm achieves an almost linear speed-up when implemented in parallel. The parallel control and shared data overheads are negligible. Even so, the best time of the PBF algorithm (that is T4 and T5) could not outperform the sequential version of the fastest method.

TABLE 5.3(a): Experimental results of the parallel Brute Force string searching algorithm (PBF) performed on the Neptune system

m	AVT1	AVT2	AVT3	AVT4	ASP2	ASP3	ASP4
6	1.188	0.617	0.426	0.334	1.926	2.787	3.560
7	1.196	0.621	0.431	0.341	1.925	2.776	3.511
8	1.196	0.622	0.430	0.337	1.922	2.782	3.547
9	1.196	0.623	0.432	0.339	1.921	2.771	3.529
10	1.207	0.626	0.433	0.431	1.928	2.788	3.542
11	1.193	0.621	0.430	0.338	1.920	2.776	3.528
12	1.194	0.623	0.431	0.339	1.917	2.770	3.522
13	1.193	0.620	0.432	0.339	1.925	2.763	3.521
14	1.187	0.619	0.428	0.339	1.918	2.775	3.505
15	1.185	0.619	0.429	0.338	1.917	2.766	3.508

TABLE 5.3(b): Experimental results of the parallel Brute Force algorithm (PBF) on the Balance 8000 system

m	AVT1	AVT2	AVT3	AVT4	AVT5	ASP2	ASP3	ASP4	ASP5
6	43.22	21.75	14.69	11.28	9.35	1.987	2.943	3.831	4.623
7	41.64	20.92	14.15	10.83	9.01	1.991	2.942	3.845	4.619
8	42.76	21.56	14.54	11.16	9.28	1.983	2.940	3.832	4.610
9	42.35	21.27	14.35	11.01	9.15	1.991	2.951	3.846	4.629
10	41.57	20.88	14.14	10.80	9.03	1.991	2.939	3.848	4.603
11	42.01	21.15	14.25	10.91	9.11	1.986	2.947	3.851	4.612
12	42.69	21.50	14.51	11.13	9.25	1.986	2.942	3.836	4.614
13	43.22	21.73	14.69	11.25	9.34	1.988	2.941	3.841	4.625
14	43.27	21.75	14.71	11.25	9.36	1.989	2.941	3.845	4.623
15	41.74	20.96	14.16	10.85	8.99	1.992	1.947	3.848	4.643

5.3.2 PARALLEL KNUTH-MORRIS-PRATT ALGORITHM (PKMP)

The sequential Knuth-Morris-Pratt string searching algorithm is proportional to the size of the string since it inspects every character of the string once and only once. If all the parallel overheads are ignored and if the average running time is assumed to be proportional to the number of character comparisons, then the average running time of the parallel KMP algorithm when performed on a single processor is:-

$$T_1 = n$$

and the average running time of the same algorithm on P processors is given by:

$$T_p = \frac{n}{P} + m - 1$$

which is exactly the number of characters in every sub-string. The average speed-up that could be achieved on a P -processor parallel computer system is:

$$S_p = \frac{T_1}{T_p} = \frac{n}{\frac{n}{P} + m - 1} = P \left(1 - \frac{P(m-1)}{n + P(m-1)} \right)$$

and the expected efficiency given by:

$$E_p = \frac{S_p}{P} = 1 - \frac{P(m-1)}{n + P(m-1)}$$

As was expected the parallel Knuth-Morris-Pratt algorithm performed very efficiently (see Tables 5.4(a) and 5.4(b)), since it achieved an almost linear speed up at a very high processor efficiency. However, even with 4 or 5 processors the parallel KMP algorithm could not outperform the sequential performance of the Boyer-Moore algorithm.

TABLE 5.4(a): Experimental results of the parallel Knuth-Morris-Pratt string search algorithm (PKMP) on the Neptune system

m	AVT1	AVT2	AVT3	AVT4	ASP2	ASP3	ASP4
6	0.737	0.392	0.277	0.224	1.882	2.661	3.295
7	0.743	0.394	0.278	0.225	1.885	2.671	3.301
8	0.747	0.396	0.281	0.226	1.887	2.657	3.298
9	0.747	0.396	0.282	0.226	1.886	2.651	3.305
10	0.753	0.401	0.284	0.227	1.879	2.654	3.320
11	0.743	0.395	0.281	0.227	1.880	2.645	3.267
12	0.746	0.396	0.282	0.226	1.882	2.643	3.307
13	0.744	0.396	0.279	0.225	1.881	2.664	3.305
14	0.741	0.394	0.278	0.226	1.879	2.667	3.279
15	0.740	0.394	0.279	0.226	1.880	2.652	3.278

TABLE 5.4(b): Experimental results of the parallel Knuth-Morris-Pratt string search algorithm (PKMP) on the Balance system

m	AVT1	AVT2	AVT3	AVT4	AVT5	ASP2	ASP3	ASP4	ASP5
6	39.11	19.66	13.23	10.05	8.19	1.989	2.956	3.891	4.777
7	37.69	18.98	12.74	9.66	7.83	1.986	2.960	3.904	4.814
8	38.71	19.43	13.09	9.96	8.07	1.992	2.958	3.887	4.799
9	38.33	19.26	12.95	9.81	7.99	1.990	2.960	3.908	4.799
10	37.66	18.95	12.68	9.68	7.80	1.987	2.971	3.892	4.828
11	37.97	19.14	12.82	9.76	7.91	1.985	2.961	3.891	4.798
12	38.55	19.38	13.05	9.89	8.07	1.989	2.954	3.900	4.780
13	39.18	19.68	13.22	10.06	8.16	1.991	2.963	3.896	4.803
14	39.07	19.63	13.17	10.02	8.13	1.990	2.965	3.898	4.804
15	37.66	18.98	12.74	9.67	7.86	1.984	2.954	3.895	4.791

5.3.3 PARALLEL KARP-RABIN ALGORITHM (PKR)

The parallel performance of the Karp-Rabin algorithm is similar to that of the Knuth-Morris-Pratt algorithm since the Karp-Rabin also has a sequential performance proportional to n . Furthermore, both algorithms were implemented in parallel using the same implementation. However, the KR method is expected to be slower than the KMP method because of the high mathematical operations involved in the computation of the hash function values. Experimentally, see Tables 2.1 and 2.2, the KR is found to be even slower than the BF. Table 5.5 reports the parallel performance of the Karp-Rabin algorithm performance on the Balance system.

TABLE 5.5: Experimental results of the parallel Karp-Rabin string searching algorithm (PKR) on the Balance system

m	AVT1	AVT2	AVT3	AVT4	AVT5	ASP2	ASP3	ASP4	ASP5
6	58.95	29.58	19.73	14.84	11.92	1.99	2.99	3.97	4.95
7	58.94	29.58	19.73	14.86	11.93	1.99	2.99	3.97	4.94
8	58.94	29.51	19.71	14.85	11.94	1.99	2.99	3.97	4.94
9	58.94	29.55	19.73	14.86	11.95	1.99	2.99	3.97	4.93
10	58.94	29.54	19.73	14.86	11.93	1.99	2.99	3.97	4.94
11	58.93	29.55	19.73	14.86	11.93	1.99	2.99	3.97	4.94
12	58.94	29.58	19.73	14.84	11.91	1.99	2.99	3.97	4.95
13	58.94	29.54	19.73	14.85	11.93	1.99	2.99	3.97	4.94
14	58.93	29.51	19.73	14.83	11.95	1.99	2.99	3.97	4.93
15	58.93	29.55	19.73	14.86	11.95	1.99	2.99	3.97	4.93

5.3.4 PARALLEL BOYER-MOORE ALGORITHM (PEM)

The Boyer-Moore string searching algorithm runs faster than all the 3 previous methods since it inspects on average only n/m characters. If the parallel running time is assumed to be proportional only to the number of character comparisons and if all the parallel overheads are neglected, then the parallel average running time of the BM algorithm when searching for all occurrences of a given pattern of length m is n/m when a single processor is being used. The parallel running time of the same algorithm performed on P processors is

$$T_p = \frac{\frac{n}{P} + m - 1}{m}$$

The average expected speed-up when P processors are used is:

$$Sp = \frac{T1}{Tp} = \frac{n}{\frac{n}{P} + m - 1}$$

which is simplified to:

$$Sp = P \left(1 - \frac{P(m-1)}{n + P(m-1)} \right)$$

and the expected efficiency given by

$$Ep = \frac{Sp}{P} = 1 - \frac{P(m-1)}{n + P(m-1)}$$

The parallel Boyer-Moore algorithm has the same expected performance as the parallel Knuth-Morris-Pratt algorithm except that the PBM is faster. The parallel performance of the Improved Boyer-Moore algorithm is exactly identical to that of the PBM for the same reasons as mentioned in the KR case. Tables 5.6(a) and 5.6(b) report the experimental results of the PBM algorithm and Tables 5.7(a) and 5.7(b) report the experimental results of the PIBM.

TABLE 5.6(a): Experimental results of the Parallel Boyer-Moore string searching algorithm (PBM) on the Neptune system

m	AVT1	AVT2	AVT3	AVT4	ASP2	ASP3	ASP4
6	0.281	0.173	0.130	0.113	1.620	2.145	2.481
7	0.239	0.151	0.117	0.104	1.579	2.025	2.281
8	0.218	0.140	0.110	0.099	1.551	1.970	2.196
9	0.207	0.135	0.108	0.095	1.532	1.912	2.159
10	0.195	0.130	0.103	0.093	1.503	1.880	2.086
11	0.181	0.123	0.100	0.091	1.474	1.807	2.000
12	0.170	0.117	0.096	0.089	1.451	1.770	1.915
13	0.170	0.119	0.095	0.089	1.423	1.782	1.908
14	0.164	0.115	0.049	0.088	1.414	1.720	1.841
15	0.155	0.111	0.092	0.086	1.394	1.688	1.805

TABLE 5.6(b): Experimental results of the Parallel Boyer-Moore string searching algorithm (PBM) on the Balance system

m	AVT1	AVT2	AVT3	AVT4	AVT5	ASP2	ASP3	ASP4	ASP5
6	4.24	2.18	1.45	1.14	1.00	1.94	2.90	3.68	4.26
7	3.55	1.82	1.26	0.97	0.81	1.95	2.81	3.66	4.39
8	2.96	1.51	1.03	0.79	0.69	1.96	2.86	3.73	4.33
9	2.63	1.37	0.93	0.73	0.62	1.91	2.82	3.63	4.26
10	2.99	1.53	1.05	0.82	0.70	1.94	2.85	3.65	4.23
11	2.26	1.17	0.83	0.63	0.56	1.94	2.73	3.59	4.09
12	2.48	1.28	0.91	0.70	0.61	1.93	2.71	3.58	4.08
13	2.16	1.13	0.81	0.60	0.53	1.92	2.67	3.60	4.07
14	2.07	1.08	0.77	0.58	0.51	1.92	2.71	3.59	4.11
15	2.00	1.04	0.73	0.55	0.49	1.92	2.74	3.62	4.15

TABLE 5.7(a): Experimental results of the Parallel Improved Boyer-Moore string searching algorithm (PBM) on the Neptune system

m	AVT1	AVT2	AVT3	AVT4	AVT5	ASP2	ASP3	ASP4	ASP5
6	4.20	2.16	1.46	1.18	0.99	1.94	2.87	3.53	4.26
7	3.50	1.79	1.23	0.98	0.81	1.95	2.83	3.58	4.29
8	2.90	1.49	1.02	0.82	0.70	1.94	2.84	3.53	4.13
9	2.59	1.33	0.95	0.73	0.63	1.95	2.74	3.56	4.15
10	2.94	1.51	1.04	0.83	0.68	1.95	2.84	3.56	4.30
11	2.20	1.15	0.79	0.62	0.52	1.91	2.79	3.56	4.28
12	2.40	1.25	0.86	0.67	0.59	1.92	2.79	3.56	4.12
13	2.13	1.10	0.75	0.60	0.52	1.93	2.83	3.56	4.12
14	1.99	1.03	0.72	0.58	0.50	1.92	2.75	3.44	4.01
15	1.94	1.01	0.70	0.56	0.48	1.92	2.77	3.48	4.06

TABLE 5.7(b): Experimental results of the Parallel Improved Boyer-Moore string searching algorithm (PBM) on the Balance system

m	AVT1	AVT2	AVT3	AVT4	ASP2	ASP3	ASP4
6	0.281	0.173	0.130	0.113	1.620	2.145	2.481
7	0.239	0.151	0.117	0.104	1.579	2.025	2.281
8	0.218	0.140	0.110	0.099	1.551	1.970	2.196
9	0.207	0.135	0.108	0.095	1.532	1.912	2.159
10	0.195	0.130	0.103	0.093	1.503	1.880	2.086
11	0.181	0.123	0.100	0.091	1.474	1.807	2.000
12	0.170	0.117	0.096	0.089	1.451	1.770	1.915
13	0.170	0.119	0.095	0.089	1.423	1.782	1.908
14	0.164	0.115	0.094	0.088	1.414	1.720	1.841
15	0.155	0.111	0.092	0.086	1.394	1.688	1.805

5.4 CONCLUSIONS

In this chapter, we first investigated 5 sequential string searching algorithms. These are the traditional Brute-Force, the Karp-Rabin, the Knuth-Morris-Pratt, the Boyer-Moore and the Improved Boyer-Moore algorithms. Empirically, it was discovered that while the BF inspects on average 1.1 characters for every character referenced, the KMP and KR inspect every character referenced once and only once [Boyer 1977]. The fastest methods are the BM and IBM since they inspect only a fraction of the string characters (n/m).

Secondly, we presented a parallel implementation for the string searching algorithms using the divide-and-conquer method. As was expected all the parallel algorithms showed a very efficient performance index on both selected MIMD type parallel computer systems (see Tables 5.8 and 5.9 below). The parallel overheads, as measured on both systems were negligible.

TABLE 5.8: Average performance of the parallel string searching methods performed on the Neptune system

Method	AVT1	AVT2	AVT3	AVT4	ASP2	ASP3	ASP4
BF	1.194	0.621	0.430	0.348	1.922	2.775	3.527
KMP	0.744	0.395	0.280	0.226	1.882	2.657	3.296
BM	0.198	0.131	0.104	0.095	1.494	1.870	2.067
IBM	0.195	0.129	0.105	0.096	1.494	1.848	2.014

TABLE 5.9: Average performance of the parallel string searching methods performed on the Balance 8000 system

Method	AVT1	AVT2	AVT3	AVT4	AVT5	ASP2	ASP3	ASP4	ASP5
KR	58.94	29.55	19.73	14.85	11.93	1.99	2.99	3.97	4.94
BF	42.45	21.35	14.42	11.05	9.19	1.99	2.94	3.84	4.62
KMP	38.39	19.31	12.97	9.85	8.00	1.99	2.96	3.87	4.78
BM	2.73	1.41	0.98	0.75	0.65	1.93	2.78	3.63	4.20
IBM	2.68	1.38	0.95	0.76	0.64	1.93	2.80	3.54	4.17

A possible extension to the work presented in this chapter, could be, for example, the search for two or more patterns simultaneously. This could be achieved by matching the required patterns in sequence and using our already developed parallel algorithms, where after finding all the occurrences of the first pattern, we pass to the second one. However such an approach would result in a time-complexity proportional to kn , where k is the number of patterns. Aho and Corasick [Aho 1975] have discovered an algorithm which is capable of achieving a time-complexity of n plus the alphabet size times the sum of the pattern lengths. Their algorithm consists of building a finite state pattern matching machine from the patterns and then using this machine to process the text string in a single pass rather than k . The pattern matching machine is a combination of three functions, a Goto, failure and an output function.

Chapter 6

PARALLEL SORTING ALGORITHMS

6.1 INTRODUCTION

The need for a good sorting algorithm is vitally important for sorting is at the core of many computer applications such as database manipulation where, very often, a list of transactions or queries have to be sorted prior to being dealt with. Sorting is basically rearranging a given set of elements S_n , where:

$$S_n = \{a_1, a_2, \dots, a_n\}$$

into some relative order. The elements a_i , ($1 \leq i \leq n$), could be a set of numbers that we wish to sort in ascending (or descending) order or a list of names that we wish to sort into alphabetical order. We have selected a set of positive numbers uniformly distributed in (0,1) for our next discussions mainly for the simple reason that sorting does not depend on the nature of the element but rather on their value.

Knuth [Knuth 1973] investigated many sorting algorithms and provided a good discussion on how the performance of any algorithm is tightly related to the experimental environment it is confronted with. Unfortunately, there is no known universal 'best' method, and one can only find a better method than the others for some particular set of conditions.

With the advent of parallel computer systems and VLSI chips the performance of these algorithms could increase considerably if a suitable parallel algorithm is designed. To date only a few parallel sorting algorithms have been implemented on MIMD computer systems. In this Chapter we have analysed the Parallel Quicksort

Algorithm (PQ) the Parallel Quicksort-Merge (PQM) and two new parallel sorting algorithms: the Parallel Bounded-Partitioned Sorting (PBPS) and the Parallel Range-Partitioned Sorting (PRPS) algorithms. Both new algorithms have shown better performance figures than the Parallel Quicksort algorithm in a chosen set of experiments.

The Quicksort algorithm has a good feature which has been exploited in the Parallel Quicksort [Evans, 1983]: at every step, the method produces 2 independent subsets which can be processed asynchronously by 2 processors. From each subset, 2 more independent subsets are created and so on. As can be seen, the parallel quicksort algorithm takes some time before all the available processors are in use. As it is, the Parallel Quicksort algorithm has a performance comparable to the Parallel Quicksort-Merge (PQM) algorithm (see Figure 6.1 comparing the theoretical speed-up of the PQ and PQM for values of P varying from 2 to 1024.

The PBPS and PRBS algorithms provide a good solution to the problem of a slow start-up of the PQ algorithm. It partitions the set to be sorted in P independent subsets which can be instantly processed by P independent processors. However a price has to be paid. The memory usage is much greater than that used by the Parallel Quicksort sort.

Another good feature of the PBPS and PRBS algorithms is that the partitioning process is carried out in parallel and has a cost almost proportional to $(n + 2 \frac{n}{p})$.

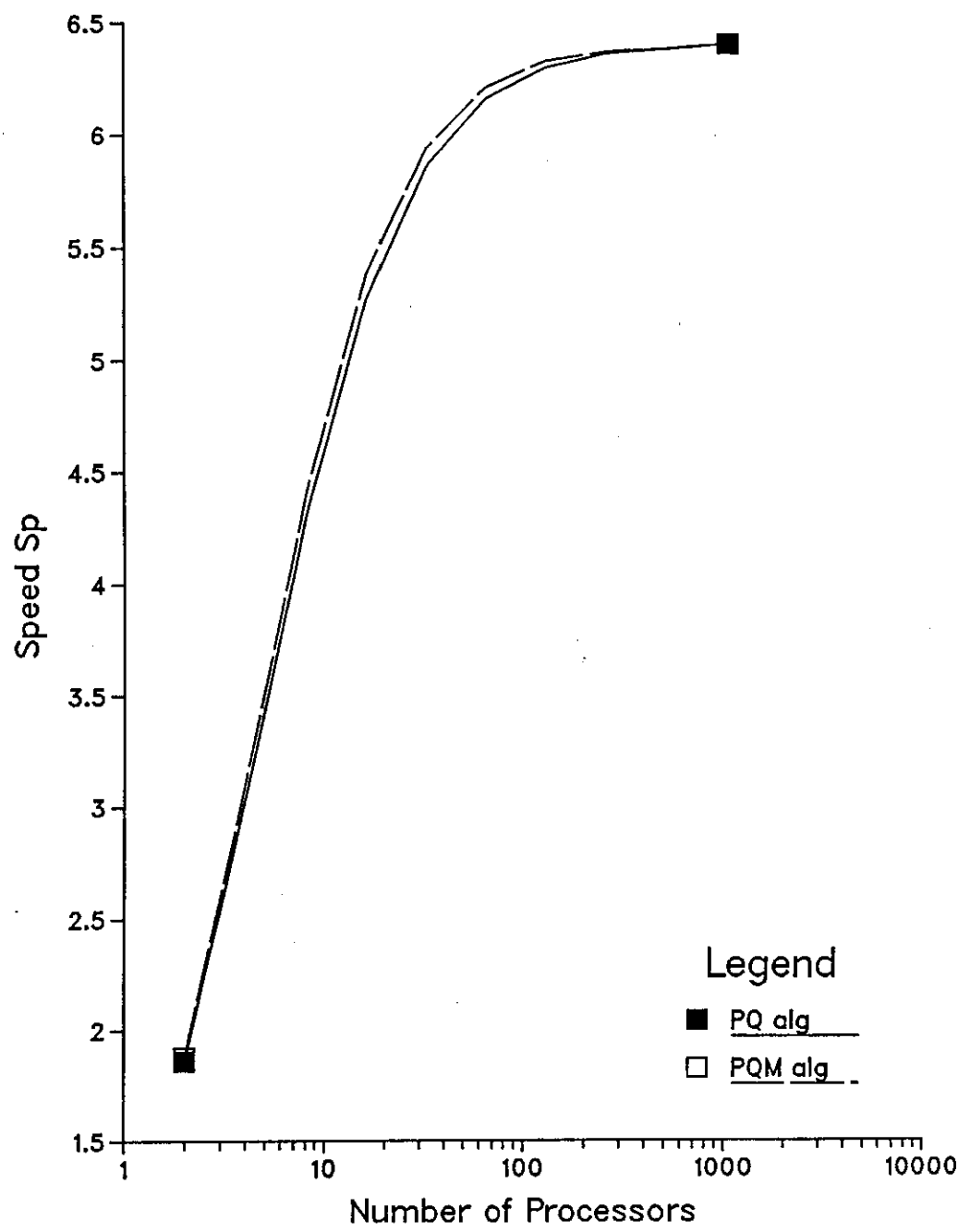


FIGURE 6.1: THEORETICAL Sp OF THE PQ AND PQM

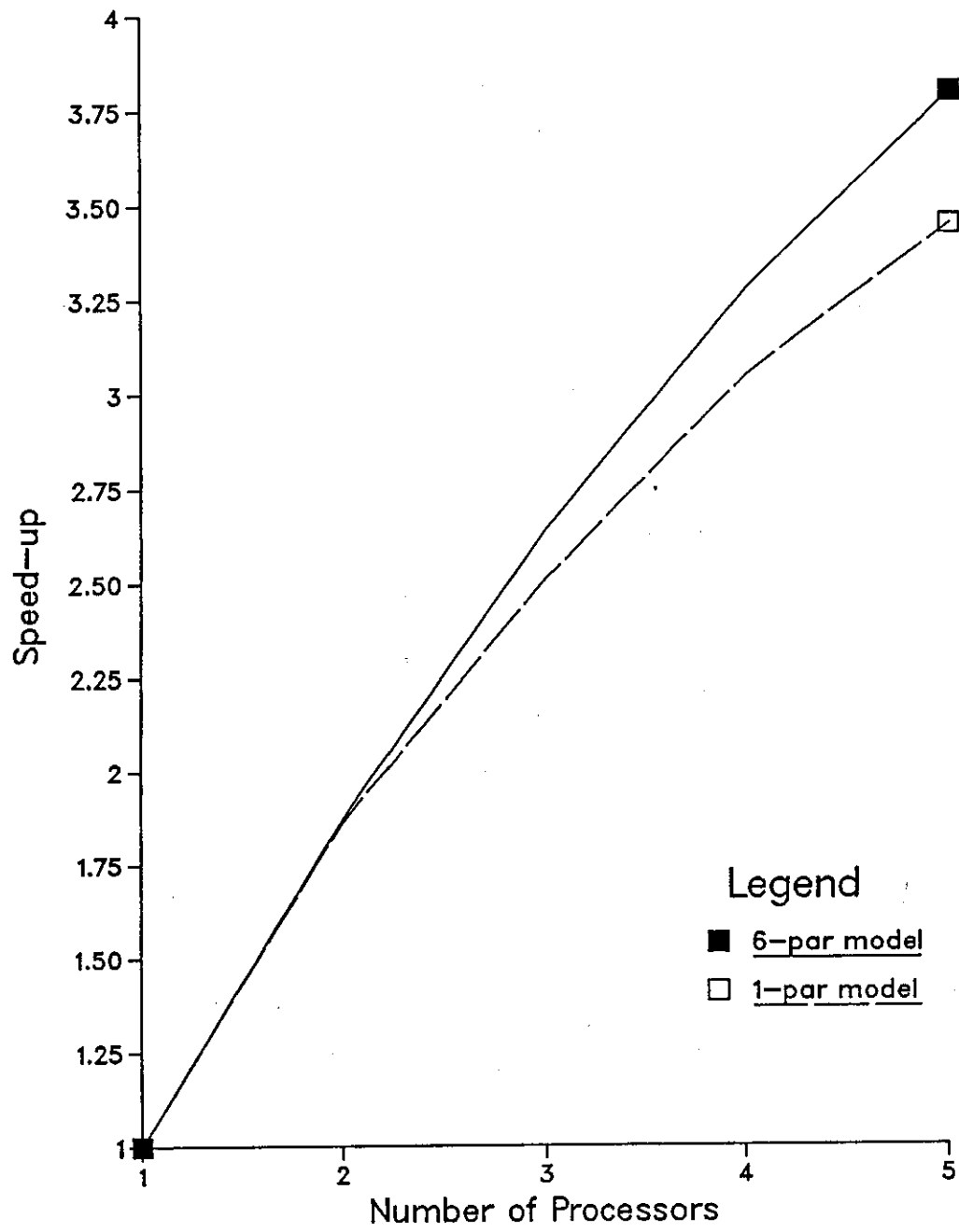


FIGURE 6.2: THEORETICAL Sp OF THE PQ

The performance analysis of the parallel sorting algorithms looked at in this Chapter, is based on the number of key comparisons*. For each algorithm 3 quantities are measured: these are the speed-up rate, the efficiency and performance factor.

In Section 6.2 the analysis is presented when the number of parallel paths (M) is equal to the number of processors (P) and in Section 6.3, we report on the case when $M > P$. Section 6.4 concludes this Chapter by summing up the major results obtained.

6.2 PERFORMANCE ANALYSIS WHEN $M=P$

Generally speaking, the set is partitioned into P subsets of nearly equal size, according to the strategy of the method. Once the P subsets are formed, they are scheduled through the P available processors to be sorted. An extra step is required only in the PQM before the set is completely sorted. The sorted subsets need to be merged together.

6.2.1 DESCRIPTION OF THE SEQUENTIAL QUICKSORT ALGORITHM

Quicksort [Hoare, 1962] or partition-exchange method which is probably more used than any other sorting algorithm is considered to be a good general-purpose algorithm. It is quite easy to implement, works well in a very large number of situations and consumes less storage than the other remaining methods [Loeser, 1974]. Hoare has

* Other parameters such as those reported in [Evans 1983] are neglected because they have no significant effect on the performance of the algorithms for large sets. (See Figures 6.6(a) and 6.6(b) comparing the theoretical speed-up of the PQ algorithms from 2 models, a 6-parameter and 1-parameter model, and from the experiments).

provided an excellent account of how Quicksort works [Hoare, 1962]. Given a set of elements, S_n , to be sorted, the method first selects at random one element, y^0 , called the partitioning element. It then rearranges the elements until the set has been partitioned into 3 parts: (a) a central part, consisting of a single element y^0 , (b) a lower (or left) subset, whose elements $(a_{11}, a_{21}, \dots, a_{n1})$, are larger than y^0 , and (c) an upper (or right) subset, $(a_{12}, a_{22}, \dots, a_{n2})$, whose elements are not smaller than y^0 . This process of placing y^0 in its proper position is known as the partitioning process. It is interesting to note that y^0 is now in its exact position and does not need to be included in any subsequent partitioning steps. The same process may be applied to each one of the 2 subsets choosing for example y^1 and y^2 as the partitioning elements for the left and right subsets respectively. A repetition of this technique eventually produces subsets containing one element or none at which point the set S_n is sorted. This process is diagrammatically presented in Figure 6.3.

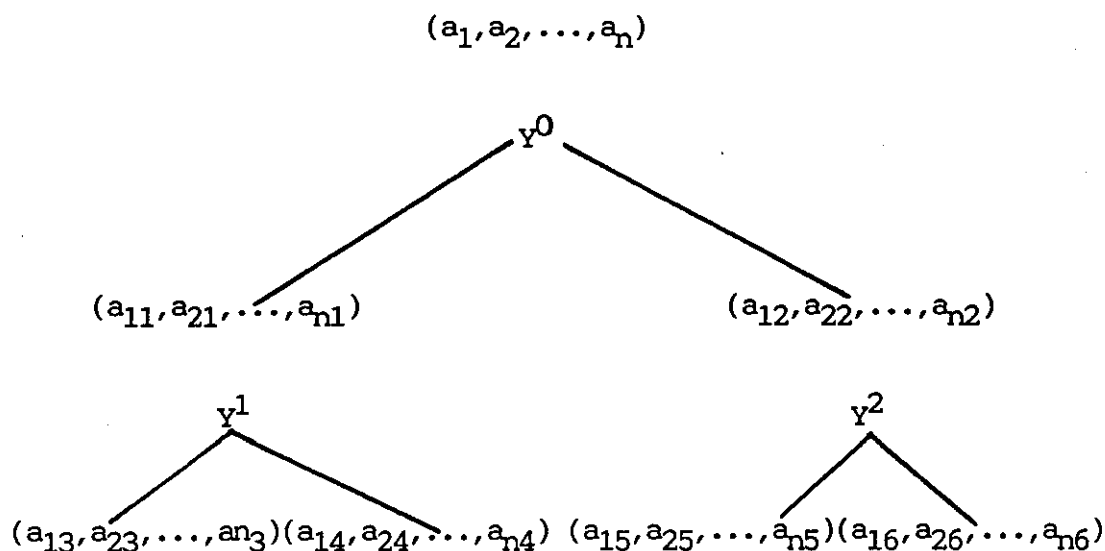


FIGURE 6.3: THE QUICKSORT ALGORITHM

Though seemingly complex, the partitioning process can be easily implemented through the following general strategy by using 2 pointers initially set to $i=l$ and $j=r-1$, where a_l and a_r are respectively the left and right-most elements, and arbitrarily choosing $Y^0 = a_r$. The scan index i is repeatedly increased by one until an element, a_i , larger than a_r is found. Then j is, in turn, continuously decreased by one until it is pointing to an element, a_j , smaller than a_r . It is obvious to see that a_i and a_j are out of place in the partitioned set, so they are exchanged. The same process of scanning from both ends and eventually exchanging elements is continued until i and j cross at which point the partitioning process is nearly complete: all that remains is to exchange a_r and the left-most element (a_i) of the right subset.

Thus, the implementation above will perform very well for many applications and it is a good general-purpose method. However, if it is used several times or used to sort large files, then it is worthwhile to implement one of the several Quicksort's variations as thoroughly investigated by Sedgewick [Sedgewick, 1975]. The main major improvements which if combined together reduce the running time of the naive Quicksort by 25%-30% [Sedgewick, 1984], can be summarised into the following points: (1) modifying the Recursive Quicksort into a non-recursive Quicksort by simulating explicitly the stack operations, (2) sorting smaller subsets (of length m) using a linear sorting method instead of involving Quicksort which exhibits high overheads with small subsets, and (3) choosing Y , the partitioning element, equals to the median of three-elements, where the three elements are the leftmost, middle and rightmost elements. Other marginal improvements include extending the median of three to the median of five (or more than five), and coding part (or the whole) of the algorithm in assembly language.

6.2.2 PARALLEL QUICKSORT ALGORITHM (PQ)

The Parallel Quicksort method consists of 3 phases (as illustrated in Figure 6.4).

Phase 1 ends when the number of active processors is exactly p , phase 2 corresponds to the situation when all the P processors are being used and the third phase ends when the number of idle processors equals P . If T_p is the run-time of the Parallel Quicksort algorithm and the duration of phase i is t_i , ($i=1,2,3$), then:

$$T_p = t_1 + t_2 + t_3 \quad 6.1$$

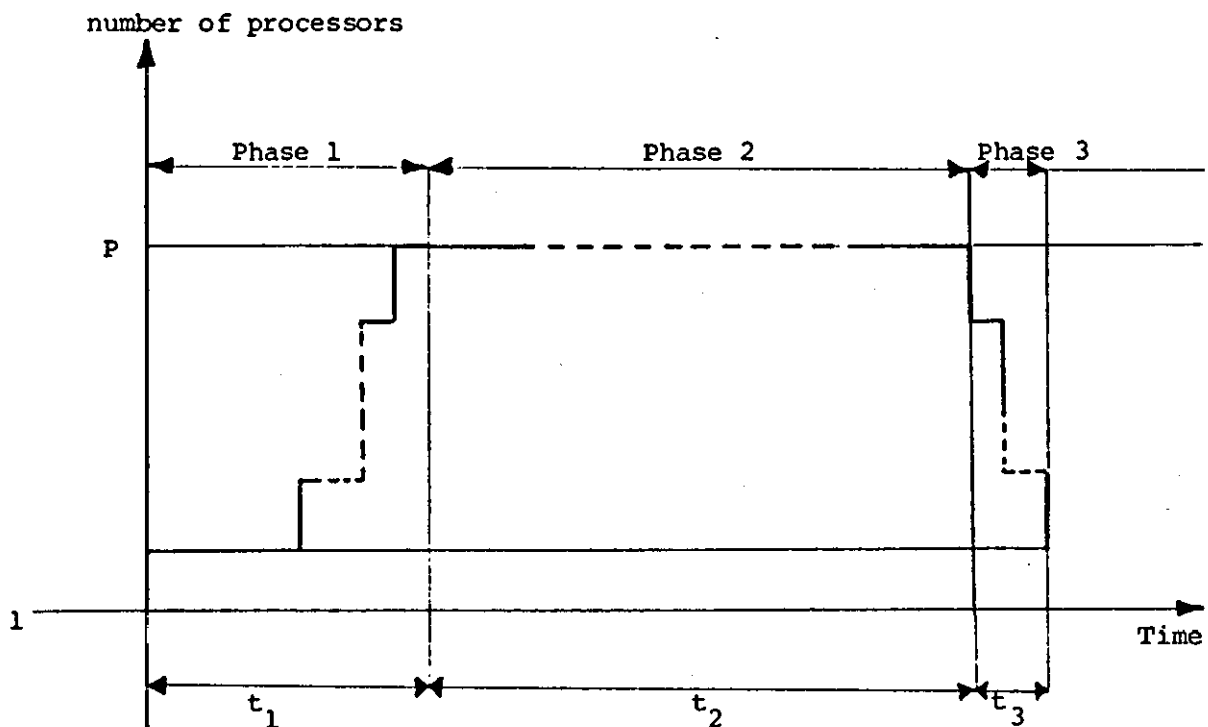


FIGURE 6.4: TIME DIAGRAM OF THE PQ ALGORITHM

However, t_3 , the time between the first processor becoming idle and the last one is difficult to be accurately estimated. Since this phase is relatively short compared to the other phases, it may be ignored. The run-time of phase 2 is estimated by:

$$t_2 = \frac{\tilde{t} - \tilde{t}_1}{p} \quad 6.2$$

where \tilde{t} is the sequential run-time of the Quicksort algorithm and \tilde{t}_1 is the sequential run-time of phase 1 of the Quicksort algorithm.

The run-time of the sequential Quicksort method, \tilde{t} , has been carefully analysed by Sedgewick [Sedgewick, 1975] by estimating the number of times each statement in the Quicksort program is executed. If we base our analysis on the number of key comparisons and apply the same technique we get:

$$\tilde{t} = \frac{12}{7} (n+1) (H_{n+1} - H_{m+2}) - 2 + (n+1) \frac{37m-94}{49(m+2)} \quad 6.3$$

(This result is from Evans [Evans, 1983] which corresponds to expression A_n , the average number of key comparisons made during the partitioning stage), where H_n is the harmonic function:

$$H_n = 1 + \frac{1}{2} + \dots + \frac{1}{n} \quad 6.4$$

To estimate \tilde{t}_1 , the sequential run-time of phase 1 of the Quicksort algorithm, we first observe that, on average, the number of key comparisons made during the first partitioning stage is $(n-1)$. (Since the partitioning element is not compared against itself). If q_i is the average length at the subsets at level i as depicted in Figure 6.5 then q_{i+1} is half the quantity (q_{i-1}) , and we have

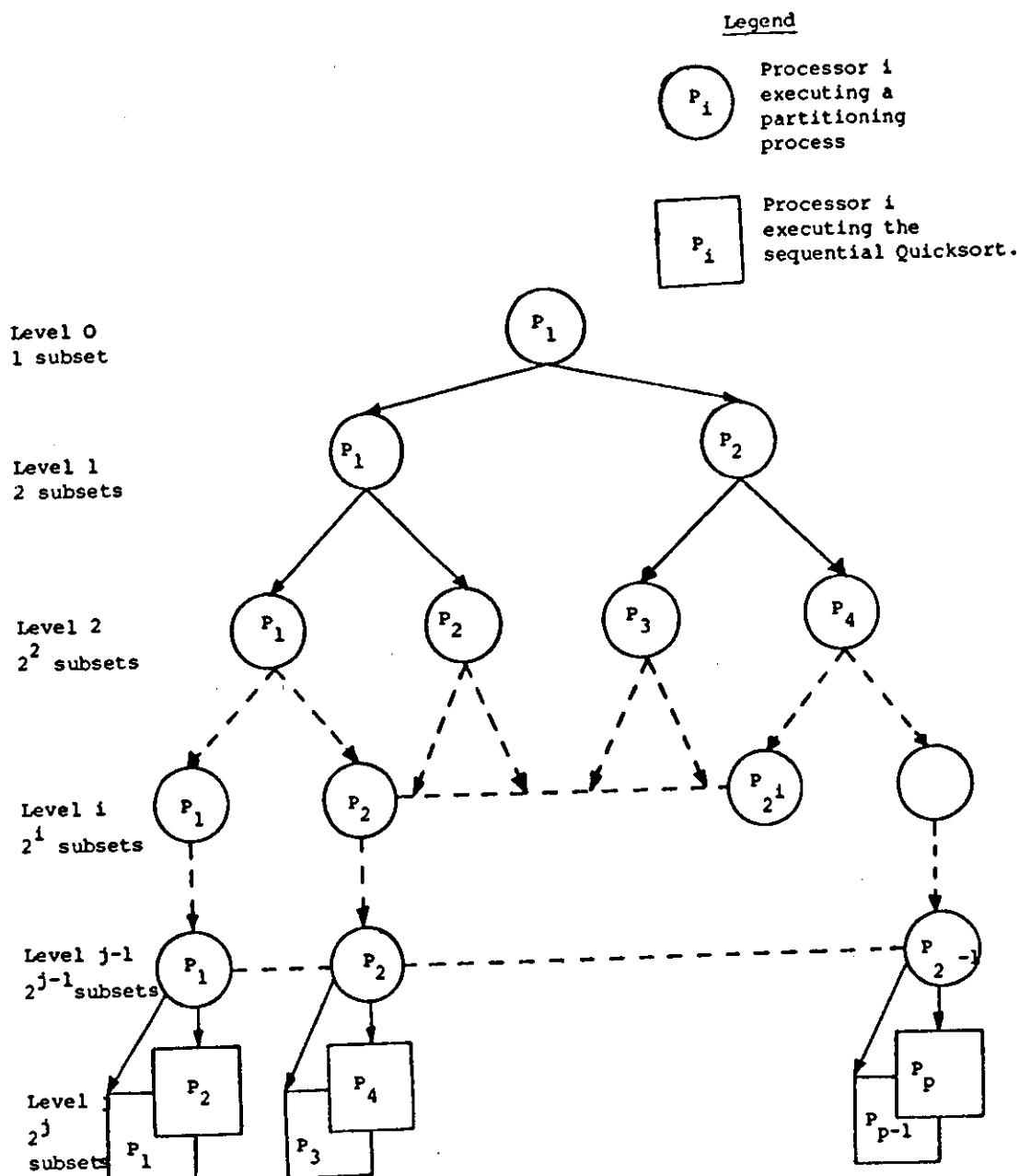


FIGURE 6.5: ALLOCATION OF PROCESSORS TO PROCESSES ($j=\log P$)

$$\begin{aligned}
 q_0 &= n \\
 q_{i+1} &= \frac{q_i - 1}{2} \quad i=1, 2, \dots, j
 \end{aligned}
 \tag{6.5}$$

where j is the log of P base 2. Using the recurrence theorem expression (6.5) is described by:

$$q_i = \frac{n+1-2^i}{2^i}, \quad i=0, 1, \dots, j \tag{6.6}$$

Each subset of the 2^i subsets obtained at level i requires on average $(q_i - 1)$ comparisons during the partitioning process. The total work, performed at level i over all the 2^i subsets, and which corresponds to the cost of partitioning at level i is given by:

$$(q_i - 1) * 2^i, \quad i=0, 1, \dots, j \tag{6.7}$$

Substituting (6.6) in (6.7) we get for the cost of partitioning 2^i subsets at level, i :

$$n+1 - 2^{i+1}$$

Phase 1 ends when all P processors are activated or when 2^j , (where $j = \log P$) subsets are created, during the sequential execution of the Quicksort algorithm. So only $j-1$ levels are required and \tilde{t}_1 is given by the sum of all level costs:

$$\tilde{t}_1 = (n+1) j + 2 - 2.2^j \tag{6.8}$$

By substituting j by $\log P$ we get the result:

$$\tilde{t}_1 = (n+1) \log P + 2 - 2P \tag{6.9}$$

Now, it remains to estimate the parallel run-time of phase 1. At any level i , ($i < 2^j$) there are always idle processors to join in the parallel partitioning process. So the 2^i subsets are processed simultaneously by 2^i processors. Once there are 2^j subsets, each processor takes a subset and performs the sequential Quicksort algorithm. So the amount of work performed by the 2^i processors is $(q_i - 1)$, ($0 \leq i \leq j$) and by summing up all these quantities we obtain:

$$t_1 = 2(n+1) (1 - 2^{-j}) - 2j \quad 6.10$$

and it is also equal to the expression below once j has been substituted by $\log P$,

$$t_1 = 2(n+1) (1 - \frac{1}{P}) - 2\log P \quad 6.11$$

Since t_3 was neglected, T_p is

$$T_p = t_1 + \frac{\tilde{t} - \tilde{t}_1}{P} \quad 6.12$$

Multiplying and dividing the left and right hand side of the above equation by P and T_p respectively we get:

$$P = \frac{P t_1 - \tilde{t}_1}{T_p} + \frac{\tilde{t}}{T_p} \quad 6.13$$

Since we have by definition:

$$S_p = \frac{\tilde{t}}{T_p} \quad 6.14$$

where S_p is the speed-up ratio of the parallel algorithm, then by substituting $\frac{\tilde{t}}{T_p}$ by S_p and rearranging the terms we get:

$$S_p = p - \frac{pt_1 - \tilde{t}_1}{T_p} \quad 6.15$$

and factorising the right hand side we obtain:

$$S_p = p(1 - \frac{pt_1 - \tilde{t}_1}{pT_p}) \quad 6.16$$

The efficiency E_p of the Parallel Quicksort method is then given by:

$$E_p = \frac{S_p}{p} = 1 - \frac{pt_1 - \tilde{t}_1}{pT_p} \quad 6.17$$

The Parallel Quicksort algorithm was performed on the Sequent Balance 8000TM to sort a randomly generated set of length 16K words. Subsets of size m or less are sorted using the Insertion sort algorithm. In all our experiments we selected $m=10$. The parallel performance of the parallel Quicksort method is given in Table 6.1 and graphically represented in Figures 6.6(a) and 6.6(b). Table 6.2 gives the predicted performance of the parallel Quicksort algorithm when P is varied from 2 to 16.

P	T_p	S_p	E_p
1	6.82	1.00	1.00
2	4.36	1.56	.78
3	4.42	1.54	.51
4	4.52	1.51	.38
5	3.67	1.86	.37

TABLE 6.1: Experimental performance of the PQ algorithm

$n = 16K, m = 10$

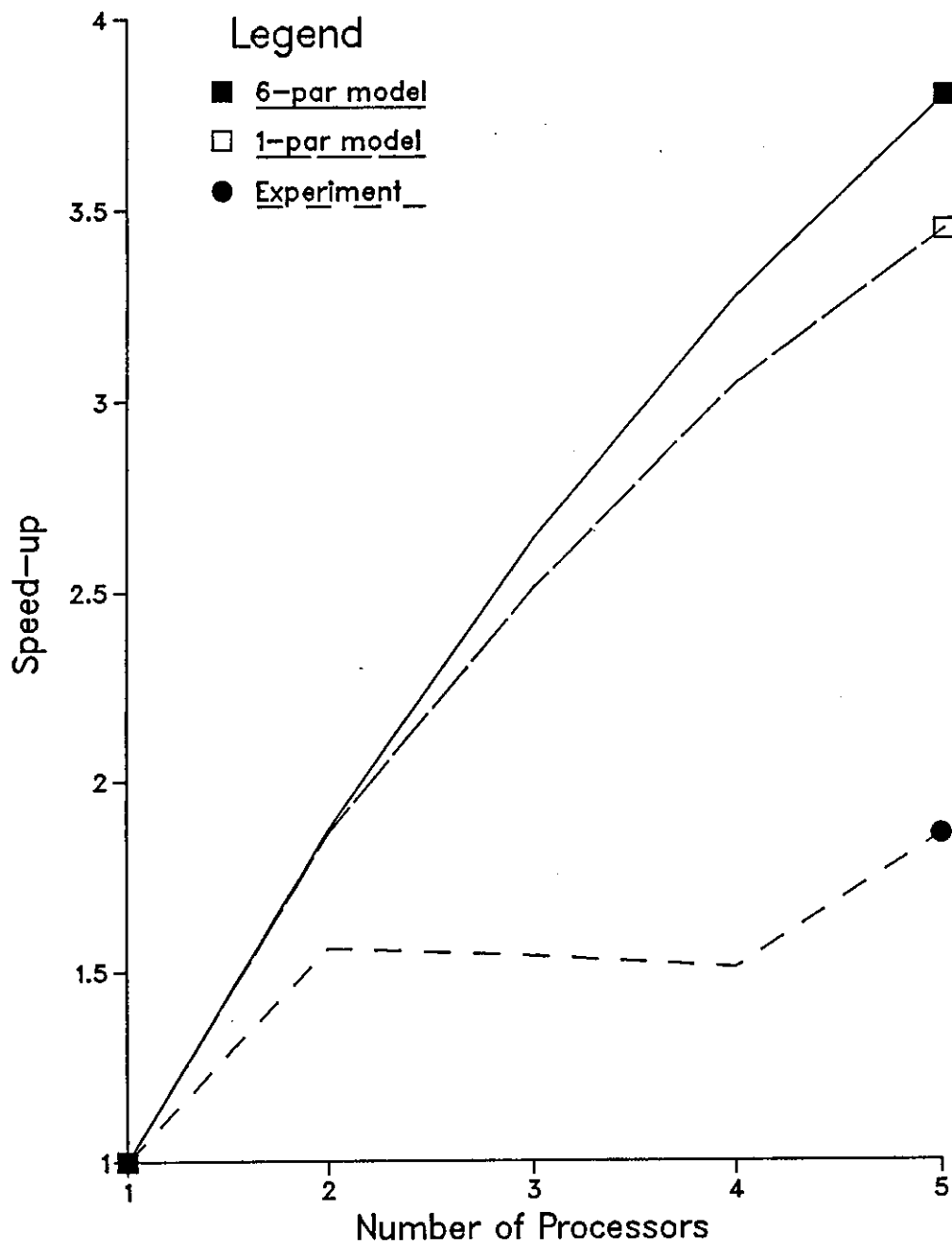


FIGURE 6.6(a): THEORETICAL AND EXPERIMENTAL Sp OF PQ

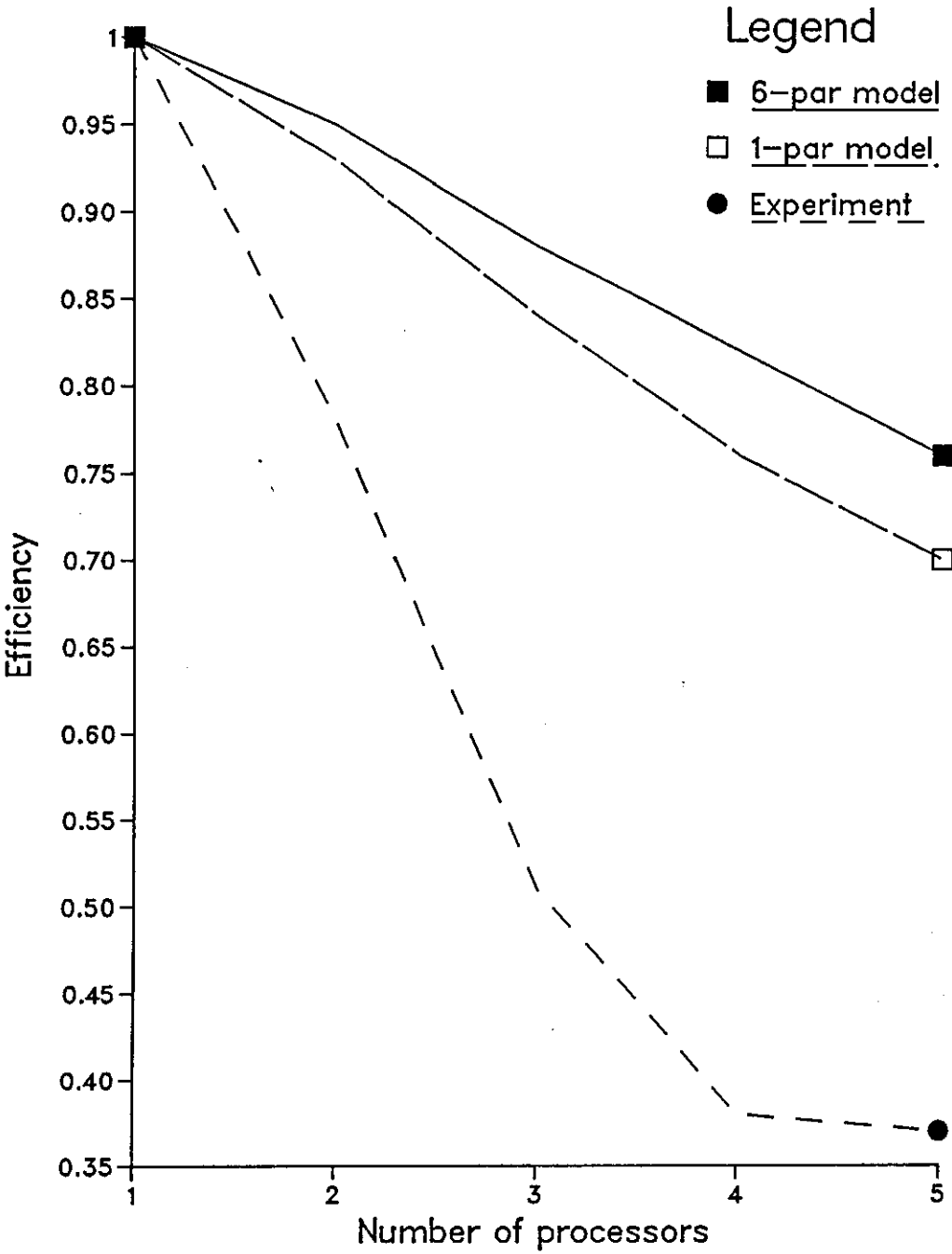


FIGURE 6.6 (b): THEORETICAL AND EXPERIMENTAL E_p OF PQ

The number of processors that achieves high Performance Factor for P processors ($PF_p = S_p * E_p$) is predicted to be six processors. However due to the fact that our analysis is overestimated, it is suggested a choice for P less than six. Experimentally we obtained the highest PF_p for 2 processors.

P	S_p	E_p	PF_p
2	1.855	.927	1.720
3	2.509	.836	2.098
4	3.046	.762	2.320
5	3.448	.690	2.378
6	3.781	.630	2.382
7	4.060	.580	2.355
8	4.299	.537	2.310
9	4.483	.498	2.233
10	4.642	.464	2.155
11	4.780	.435	2.077
12	4.902	.409	2.003
13	5.101	.385	1.931
14	5.107	.365	1.863
15	5.194	.346	1.798
16	5.272	.330	1.737

TABLE 6.2: Theoretical performance of the Parallel Quicksort Algorithm (median of the three versions), $n=16K$ and $m=10$

6.2.3 PARALLEL QUICKSORT MERGE (PQM)

The main purpose of the analysis of the Parallel Quicksort-Merge algorithm is to provide an easy and known alternative performance analysis to the parallel Quicksort algorithm. Intuitively, phase 1 of the PQ algorithm and the merging phase of the PQM are of the same order (if they are not equal). If we assume that both algorithms have the same run-time then as a validation procedure for the analysis of the PQ algorithm, the establishment of the analytical performance of the PQM could make a good approximation for the PQ. Theoretically, the PQ and PQM, have almost equal parallel performance for the set of assumptions made. (See Figure 6.1 and Tables 6.1 and 6.3).

P	T_{pc}	T_{pe}	S_{pc}	S_{pe}	E_{pc}	E_{pe}
1	6.78	6.80	1.00	1.00	1.00	1.00
2	4.44	4.50	1.53	1.51	.76	.76
4	3.40	3.41	1.99	1.99	.50	.50

TABLE 6.3: Experimental performance of the PQM algorithm
n=16K, m=10

The PQM algorithm consists of two phases: (a) a sorting phase where each processor sorts a subset of length $(\frac{n}{p})$ using the sequential Quicksort, and (b) a merging phase to obtain a sorted set by merging the P sorted subsets. If T_{PS} and T_{PM} are the run-time of respectively the sorting and merging phased then, T_p , the overall parallel run-time is expressed by:

$$T_P = T_{PS} + T_{PM} \quad 6.18$$

Since each subset of size $(\frac{n}{p})$ is sorted using a sequential version of the Quicksort algorithm, then T_{PS} is the same as (defined in 6.3), except that n is substituted by $(\frac{n}{p})$. So T_{PS} is

$$T_{PS} = \frac{12}{7} \left(\frac{n}{p} + 1 \right) + \left(H \frac{n}{p} + 1 - H_{m+2} \right) + 2 + \left(\frac{n}{p} + 1 \right) \frac{37m-94}{49(m+2)} \quad 6.19$$

Once all the P subsets are individually sorted, they are merged using a parallel merging method. The choice of a particular merging algorithm is irrelevant because any algorithm used to merge 2 sorted files of the same length (n) by comparison of keys does at least $(2n-1)$ such comparisons [Baase, 1983]. The two-way merge was selected since it is less complicated to implement than the alternative methods.

At the start of the merging phase, every 2 neighbouring subsets are merged to form a new sorted subset of size $(2 \frac{n}{p})$. The number of such processes is then $(\frac{P}{2})$. During the following step only $(\frac{P}{4})$ processors are being used. The same process is repeated until only one processor is used to merge 2 sorted subsets, each containing exactly $(\frac{n}{2})$ elements and produces the final sorted set S_n . The merging process is diagrammatically shown in Figure 6.7 when $P=8$. The notation S_iMS_{i+1} is used to name the set obtained by merging S_i and S_{i+1} .

In this analysis, we assume n and p to be a power of 2. The parallel merge phase can be completed in only $\log P$ steps. Merging at one step is performed in parallel and control is passed to the next step if, and only if, all the merging processes have been completed.

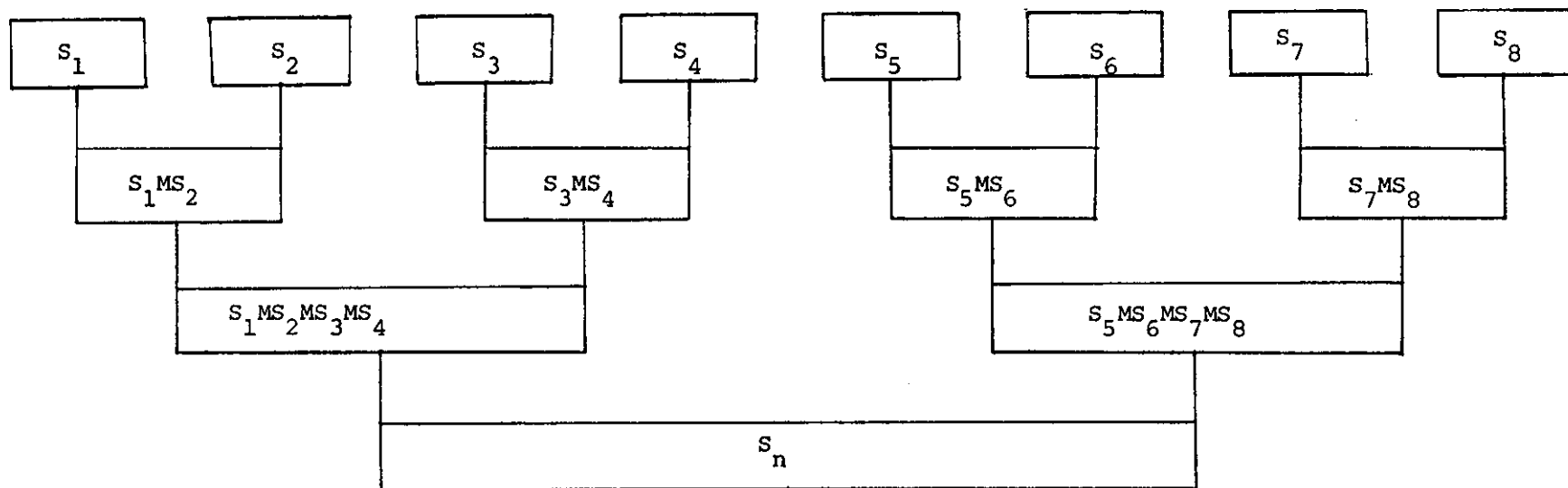


FIGURE 6.7: PARALLEL MERGING PROCESS ($P = 8$)

At step i , there are $\frac{P}{2^i}$ subsets of size $(\frac{n}{p})2^i$ each. T_{PM} , the run-time of the parallel merging phase is then:

$$T_{PM} = \sum_{i=0}^{j-1} c_i \quad 6.20$$

where $j = \log P$ and c_i is the cost of step i . c_i is simply,

$$c_i = 2(\frac{n}{p} \cdot 2^i) - 1 = \frac{n}{p} \cdot 2^{i+1} - 1 \quad 6.21$$

Substituting (6.21) in (6.20) and summing up, we get T_{PM}

$$T_{PM} = 2 \frac{n}{p} (2^j - 1) - j \quad 6.22$$

which is also equal to,

$$T_{PM} = 2n (1 - \frac{1}{p}) - \log P \quad 6.23$$

after substituting j by $\log P$.

S_p the speed-up ratio of the PQM algorithm is obtained as,

$$S_p = \frac{T_1}{T_p} = \frac{\hat{t}_1}{T_{PS} + T_{PM}} \quad 6.24$$

and the efficiency,

$$E_p = \frac{S_p}{p} \quad 6.25$$

6.2.4 PARALLEL PARTITIONED SORTING ALGORITHMS (PPS)

The motivation which led to the development of 2 parallel partitioned sorted algorithms (the bounded and range partitioned sorting algorithms) was how to overcome the disadvantage observed in the phase 1 of the PQ method. We first observed empirically that if somehow the original set is partitioned into P subsets such that all elements of subsets S_i are not greater than any element of subset S_{i+1} (right subset for S_i) and not smaller than any elements of subset S_{i-1} (left subset for S_i). In other words, if a parallel partitioning process could be found such that the original set is partitioned not only into 2 independent subsets as in the PQ but into P independent subsets, then phase 1 of the PQ method could be totally eliminated and hence linearly improve the overall performance. Here again the ideal situation is to produce P subsets of nearly equal sizes.

Both PPS methods are based on first defining an array $U(1:P+1)$ such that

$$U_1 < U_2 < \dots < U_{p+1}$$

The parallel Bounded-Partitioned sorting method (PBPS) selects $U_1=S(1)$, $U_2=S(\frac{n}{p})$, ..., $U_p=S((p-1)\frac{n}{p})$ and $U_{p+1}=S(n)$ and then sorts the array U in ascending order. The second method, the Parallel Range-Partitioned sorting algorithm (PRPS) is used only if the range of the set S_n is known and the array U is selected as shown below:

$$U_i = a + (i-1) \frac{b-a}{p}, \quad i=1, p+1 \quad 6.26$$

where a and b are the lower and upper bounds of S_n . It is easy to show that U_i is already ordered in ascending order by expressing U_{i+1} as a function of U_i

$$U_{i+1} = a + i \frac{b-a}{p}, \quad i=0, p \quad 6.27$$

which is also equal to,

$$U_{i+1} = a + (i-1) \frac{b-a}{p} + \frac{b-a}{p}, \quad i=0, p \quad 6.28$$

replacing $a + (i-1) \frac{b-a}{p}$ by U_i we get

$$U_{i+1} = U_i + \frac{b-a}{p} \quad 6.29$$

The above expression shows that,

$$U_{i+1} > U_i, \quad \text{for } i=0, p \quad 6.30$$

since $\frac{b-a}{p} > 0$ and hence the sorting of U is avoided in the PRBS method.

Once the bound array U has been defined, all P processors are activated. If processors are numbered from 1 to P , then processor i_p in the PBPS algorithm picks all elements a_i such that:

$$a_i \leq U(2), \quad \text{if } i_p = 1$$

$$a_i > U(p), \quad \text{if } i_p = p \quad i = 1, n$$

$$U(i_p) < a_i \leq U(i_p+1) \text{ otherwise} \quad 6.31$$

The PRPS method makes every processor number i_p , pick all elements from the set S_n which are,

$$U(i_p) < a_1 \leq U(i_p+1) \quad 6.32$$

The elements picked by every processor are first stored in a local array. They are copied back to the original array only when all the sorting processes have been completed. The exact start index for every processor is saved in an index table. The partitioning process and writing back to the original set has a run-time proportional to $(n + \frac{n}{p})$ and the sorting was defined previously in 6.19. So T_p is

$$T_p = n + \frac{n}{p} + \frac{12}{7} \left(\frac{n}{p} + 1 \right) \left(H \frac{n}{p} + 1 + H_{m+2} \right) + 2 + \left(\frac{n}{p} + 1 \right) \frac{37-94}{49(m+2)} \quad 6.33$$

First we compared the theoretical speed-up of the PQ and PPS algorithms for P varying from 2 to 16 (see Figure 6.8). As predicted, the performance of the PPS algorithms start outperforming the Parallel Quicksort algorithms as P increases from 3-4, then it shows clearly that for $P > 4$, the PPS greatly outperformed the PQ algorithm. Figure 6.9 which plots the experimental results given in Tables 6.1, 6.4 and 6.5 of the PQ, PBPS and PRPS algorithms confirm the theoretical results of these algorithms. However, PRPS is faster and more efficient than the PBPS method. This is mainly because of the fact that PRPS creates subsets of nearly equal size than the PBPS algorithm does. Figure 6.10 shows a sample of subset sizes created by the 2 different PPS algorithms when $P=64$ and n , the set size, equals to 16K words.

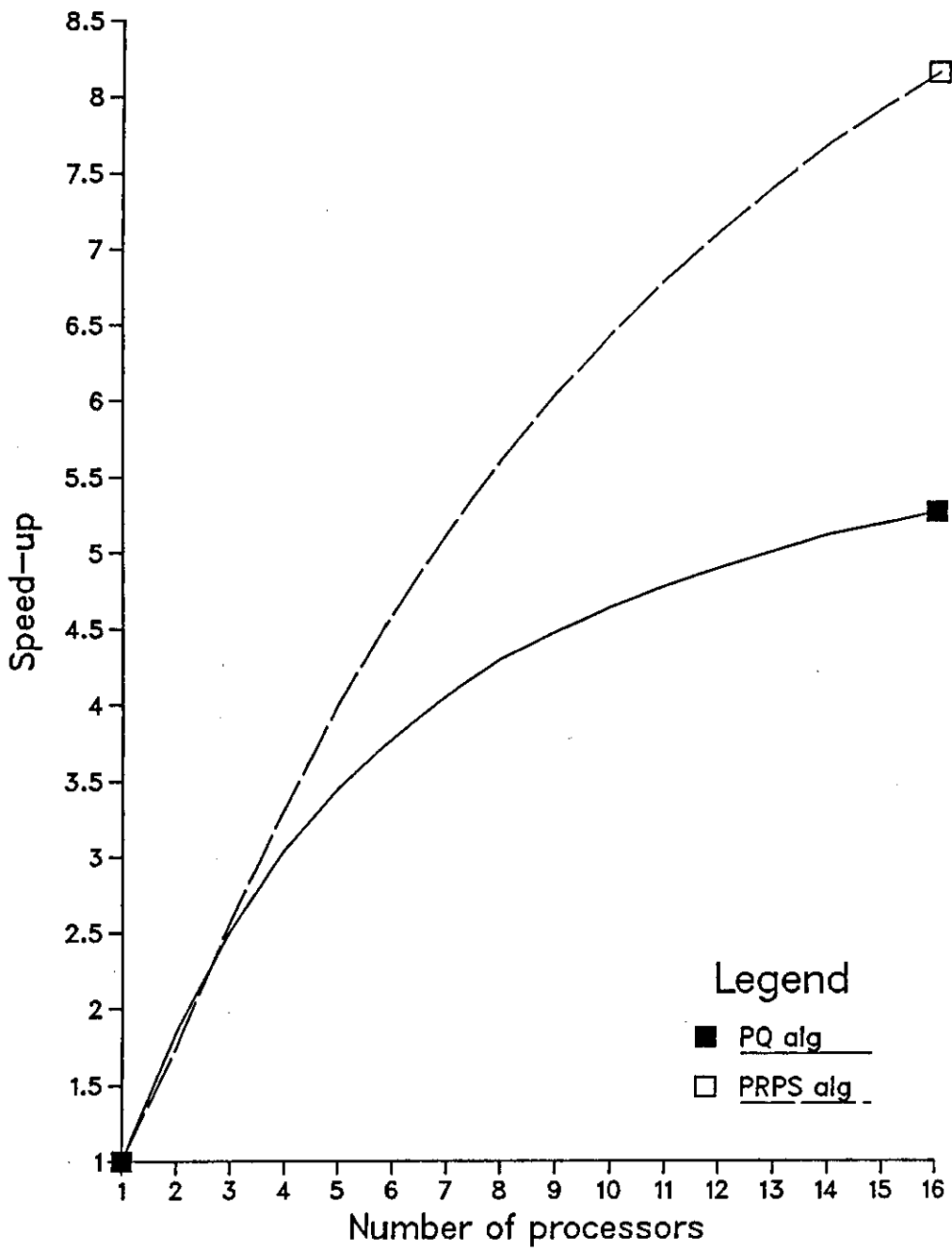


FIGURE 6.8: THEORETICAL S_p OF THE PQ AND PPS

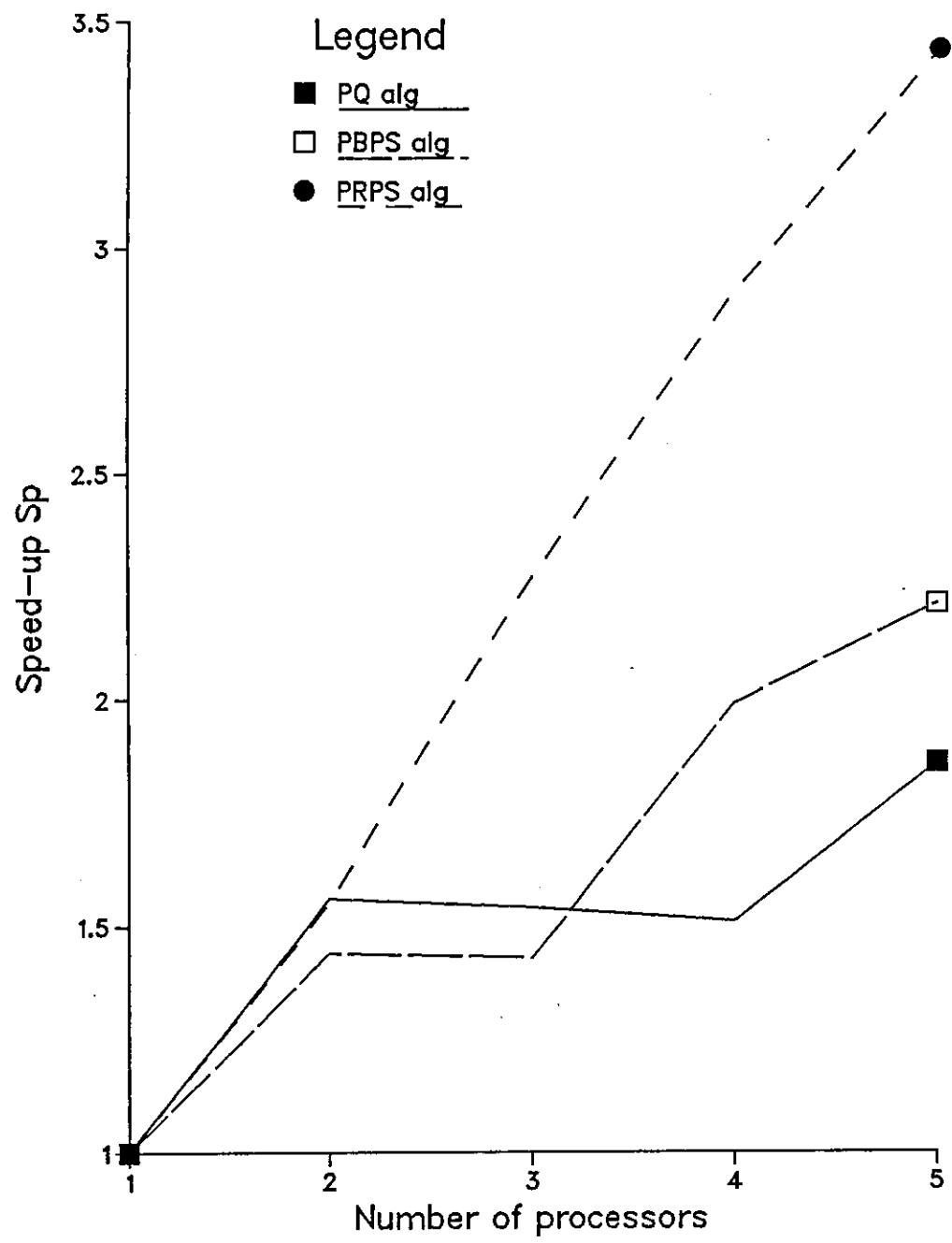


FIGURE 6.9: EXPERIMENTAL Sp OF THE PQ, PBPS AND PRPS

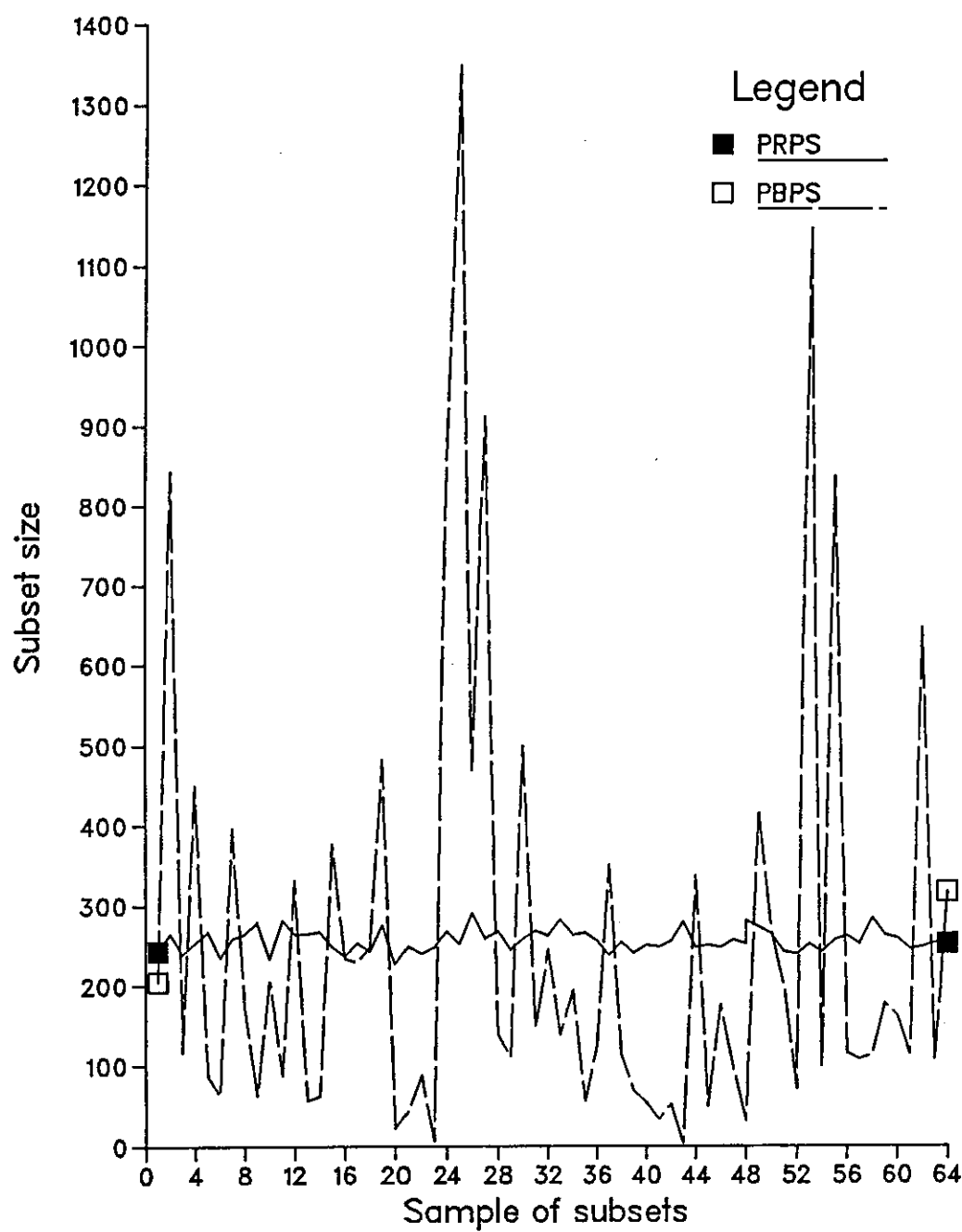


FIGURE 6.10: SUBSET SIZES CREATED BY PBPS AND PRPS

P	T _{pc}	T _{pe}	S _{pc}	S _{pe}	E _{pc}	E _{pe}
1	6.78	6.82	1.00	1.00	1.00	1.00
2	4.20	4.41	1.61	1.55	.81	.77
3	2.89	3.01	2.35	2.27	.78	.76
4	2.26	2.35	3.00	2.90	.75	.73
5	1.86	1.98	3.65	3.44	.73	.69

TABLE 6.4: Experimental performance of the PRPS algorithm
n = 16K, m = 10

P	T _{pc}	T _{pe}	S _{pc}	S _{pe}	E _{pc}	E _{pe}
1	6.78	6.82	1.00	1.00	1.00	1.00
2	4.51	4.73	1.50	1.44	.75	.72
3	4.65	4.76	1.46	1.43	.49	.48
4	3.37	3.42	2.01	1.99	.50	.50
5	3.00	3.08	2.26	2.21	.45	.44

TABLE 6.5: Experimental performance of the PBPS algorithm
n = 16K, m = 10

6.3 PERFORMANCE ANALYSIS WHEN M>P

One of the methods used in efficient parallel algorithms development is load balancing. That is keeping all the processors busy performing some useful work. Load balancing can usually be best achieved by splitting the task into P parallel subtasks of nearly equal execution time. Although all the processors may not complete

execution at the same time, it is hoped that the time between the first and last processor becoming idle is negligible. Failing to do so, results in a poor performance. Some problems naturally divide into P equal subtasks, for example, matrix and vector manipulations. However, there are some types of problems, for example some graph algorithms, which are difficult to balance on a P -MIMD parallel computer system. In this case we notice that one processor takes the whole junk of work leaving the others idle most of the time.

A solution to this problem which sometimes improves the parallel performance of some algorithms is to divide the task into a number of subtasks (M) larger than P and schedule them for execution through the P processors. As number of subtasks increases, their execution time decreases and most importantly the processors are more balanced.

In the following 3 sub-sections we reanalyse the 4 parallel sorting algorithms when M , the number of subtasks, is greater than P . The words "subsets" and "subtasks" will be used to mean the same thing and M should not be confused with m , the size of the largest subset that is sorted using the linear insertion sort algorithm.

6.3.1 PARALLEL QUICKSORT ALGORITHM (PQ)

Through parallel partitioning, as in phase 1 of Figure 6.4, M subsets are created, then they are scheduled to be executed by the P processors. Using a sequential version of the Quicksort, each processor sorts $(\frac{M}{P})$ subsets each containing on average $(\frac{n}{M})$ elements. If T_p is the run-time of the parallel Quicksort algorithm then we have:

$$T_p = t_1 + \left(\frac{M}{P}\right) \frac{\tilde{t} - \tilde{t}_1}{M} \quad 6.34$$

which simplifies to

$$T_p = t_1 + \frac{\tilde{t} - \tilde{t}_1}{P} \quad 6.35$$

where t_1 is the run-time of the parallel partitioning process up to the point where M subsets are obtained, \tilde{t}_1 is the run-time of the same above-mentioned process but performed sequentially and \tilde{t} is the sequential Quicksort run-time. \tilde{t} is exactly the same as defined in 6.3 and \tilde{t}_1 is the same as defined in 6.9 except that P is substituted by M

$$\tilde{t}_1 = (n+1) \log M + 2-2M \quad 6.36$$

M is assumed to be a power of 2 for convenience only.

t_1 is now calculated. Up to level $j' = \log P$, there are sufficient processors to execute the $2^{j'}$ subsets. So the run-time of the parallel partitioning process up to level j' is

$$\sum_{i=0}^{j'} c_i \quad 6.37$$

where c_i is the cost of level i (i.e. the cost of partitioning 2^i subsets in parallel) and is given by,

$$c_i = q_i - 1$$

where q_i was defined previously in equation 6.6. As i increases from j' to $j = \log M$, more subsets are created than the available number of processors. Therefore the run-time of the parallel

partition process of $2P$ subsets until M subsets are created is,

$$\sum_{i=j'+1}^{j-1} \left(\frac{2^i}{P}\right) c_i \quad 6.38$$

Every processor has to partition $\left(\frac{2^j}{P}\right)$ subsets and t_1 is the sum of 6.37 and 6.38, which is,

$$t_1 = \sum_{i=0}^{j'} c_i + \sum_{i=j'+1}^{j-1} \left(\frac{2^i}{P}\right) c_i \quad 6.39$$

This is also equal to the following equation after inserting the expression of c_i , and rearranging the terms of the sum,

$$\sum_{i=0}^{j'} c_i = (n+1) (2 - 2^{-j'}) - 2(j'+1) \quad 6.40$$

Repeating the same operation on 6.38 we get,

$$\sum_{i=j'+1}^{j-1} \left(\frac{2^i}{P}\right) c_i = \left(\frac{n+1}{P}\right) (j-j'-1) - \frac{2}{P} (2^j - 2 \cdot 2^{j'}) \quad 6.41$$

Summing up 6.40 and 6.41 we have,

$$t_1 = (n+1) \left(\frac{j-j'-1}{P} + 2 - 2^{-j'}\right) - 2 \left(\frac{2 - 2 \cdot 2^{j'}}{P} + j'+1\right) \quad 6.42$$

which is also equal to

$$t_1 = (n+1) \left(\frac{1}{P} \log \left(\frac{M}{P}\right) + 2 - \frac{2}{P}\right) - 2 \left(\frac{M}{P} + \log P - 1\right) \quad 6.43$$

after substituting j' and j by their respective expressions.

From equation 6.35 we get,

$$T_p = t_1 - \frac{t_1}{P} + \frac{\tilde{t}}{P} \quad 6.44$$

where $\frac{\tilde{t}}{P}$ is not a function of M. Now let us further simplify the expression,

$$\begin{aligned} t_1 - \frac{\tilde{t}}{P} &= (n+1) \left(\frac{1}{P} \log \frac{M}{P} + 2 - \frac{2}{P} \right) - 2 \left(\frac{M}{P} + \log P - 1 \right) \\ &\quad - \left(\frac{(n+1) \log M + 2 - 2M}{P} \right) \end{aligned}$$

The terms in M are eliminated and we get,

$$t_1 - \frac{\tilde{t}}{P} = \frac{1}{P} [(n+1)(2P - \log P - 2) + 2P(1 - \log P) - 2] \quad 6.45$$

which we let equal to,

$$t_1 - \frac{\tilde{t}}{P} = \frac{1}{P} \Delta t_1 \quad 6.46$$

So the speed-up of the parallel Quicksort which is not a function of M, is

$$S_p = \frac{\tilde{t}}{T_p} = \frac{\tilde{t}}{\frac{\Delta t_1 + \tilde{t}}{P}} = P \frac{\tilde{t}}{\tilde{t} + \Delta t_1} \quad 6.47$$

The efficiency is then,

$$E_p = \frac{S_p}{P} = \frac{\tilde{t}}{\tilde{t} + \Delta t_1} \quad 6.48$$

The theoretical and experimental results of the Parallel Quicksort when the number of paths is greater than P are listed respectively in Tables 6.6 and 6.7.

M	P	S_p	E_p	PF_p
8	2	1.855	.927	1.720
	3	2.551	.850	2.169
	4	3.046	.762	2.320
	5	3.448	.690	2.378
16	2	1.855	.927	1.720
	3	2.594	.865	2.243
	4	3.046	.762	2.320
	5	3.519	.704	2.477
32	2	1.855	.927	1.720
	3	2.638	.879	2.320
	4	3.046	.762	2.320
	5	3.593	.719	2.583
64	2	1.855	.927	1.720
	3	2.684	.895	2.402
	4	3.046	.762	2.320
	5	3.671	.734	2.695

TABLE 6.6: Theoretical results of the PQ when $M > P$
 $n = 16K$ words and $m = 10$

M	P	T_p	S_p	E_p
8	2	3.94	1.73	.86
	3	3.57	1.90	.63
	4	3.50	1.94	.49
	5	3.54	1.92	.38
16	2	3.89	1.75	.87
	3	3.00	2.27	.76
	4	2.59	2.63	.66
	5	2.60	2.62	.52
32	2	3.95	1.72	.86
	3	2.99	2.27	.76
	4	2.65	2.57	.64
	5	2.66	2.56	.51
64	2	3.86	1.76	.88
	3	2.99	2.27	.76
	4	2.53	2.69	.67
	5	2.39	2.85	.57
128	2	3.92	1.73	.87
	3	2.97	2.29	.76
	4	2.58	2.64	.66
	5	2.34	2.91	.58
256	2	3.96	1.72	.86
	3	3.10	2.19	.73
	4	2.65	2.57	.64
	5	2.40	2.83	.57

TABLE 6.7: Experimental performance of the PQ when $M > P$
 $n = 16K$ words and $m = 10$

The experimental performance increases as M increases. It starts decreasing for $M = 256$ and upwards. This shows that the parallel control overheads become significant as M increases from 256. The theoretical shows a constant increase in the Parallel Quicksort performance as M increases.

6.3.2 PARALLEL QUICKSORT-MERGE ALGORITHM (PQM)

The original set is divided into M subsets, each containing $(\frac{n}{M})$ elements. Each processor sorts $(\frac{M}{P})$ subsets and T_{PS} the parallel run-time for sorting these subsets on a p -MIMD multiprocessor system is equal to $(\frac{M}{P})$ times (t_s) the sequential run-time of the Quicksort algorithm to sort a set of length $(\frac{n}{M})$ and it is given by,

$$T_{PS} = \frac{1}{P} \left[\frac{12}{7} (n+p) \left(H_{\frac{n}{M}+1} - H_{m+2} \right) + 2M + (n+p) \frac{37m-94}{49(m+2)} \right] \quad 6.49$$

The analysis of the merging process is similar to that used in Section 6.2.3. For the purpose of this analysis we assume P and M power of 2 and we define the following quantities:

- s_i : number of subsets at level i if the merging process
- q_i : average subset length at level i
- c_i : cost of merging 2 neighbouring subsets at level i .

It is obvious that there is a set at level 0 containing n elements, and the final merging (at level 1) of the 2 subsets of size $\frac{n}{2}$ each is performed after $n-1$ key comparisons. Therefore we have as initial conditions,

$$s_0 = 1$$

$$q_0 = n \quad 6.50$$

$$c_1 = n-1$$

We noticed that as the level number increases by one, the number of subsets is doubled and the average subset length is halved,

$$s_{i+1} = 2s_i, \quad 1 \leq i \leq j, \quad j = \log P \quad 6.51$$

$$q_{i+1} = q_i/2, \quad 1 \leq i \leq j$$

By using the recurrence theorem we derive the general terms of q_i and s_i

$$q_i = \frac{n}{2^i}, \quad i = 0, j \quad 6.52$$

$$s_i = 2^i, \quad i = 0, j$$

The run-time of merging 2 sorted subsets at level i is

$$c_i = 2q_i - 1 \quad 6.53$$

and it is also equal to the next expression when q_i is replaced by its value from equation 6.52

$$c_i = 2 \frac{n}{2^i} - 1 \quad 6.54$$

Table 6.8 below summarises the variations of the quantities s_i , q_i and c_i when i varies from 0 to j ,

Level i	No. of Subsets s_i	Size of Subset q_i	Run-time of a Merge, c_i
j	$2^j = M$	$\frac{M}{2^j} = \frac{n}{M}$	$2 \frac{n}{M} - 1$
$j'+1$	$2^{j'+1} = 2p$	$\frac{n}{2^{j'+1}} = \frac{n}{2p}$	$2 \frac{n}{p} - 1$
\vdots	\vdots	\vdots	\vdots
1	2	$\frac{n}{2}$	$n-1$
0	1	n	No merging

TABLE 6.8

From Table 6.8 we distinguish two stages depending on the number of active processors since the number of merging paths varies by two-fold: (a) when P (or less) processors are used. This stage corresponds to level i , ($1 \leq i \leq j'+1$), and (b) when the number of merging paths exceeds P and this is for levels i ($j' + 2 \leq i \leq j$).

The merging phase (a) has a run-time equal to,

$$t_a = \sum_{i=1}^{j'+1} c_i \quad 6.55$$

which is also equal to,

$$t_a = n(2 - 2^{j'}) - (j' + 1) \quad 6.56$$

At any level i of stage (b), every processor performs $(s_i/2P)$ merging paths and t_b , the run-time of stage (b) is expressed as,

$$t_b = \sum_{i=j'+2}^j \left(\frac{s_i}{2P} \right) c_i \quad 6.57$$

which is also equivalent to,

$$t_b = \frac{n}{p} (j - j' - 1) - \frac{1}{p} (2^j - 2 \cdot 2^{j'})$$

The cost of the parallel merging algorithm is then,

$$T_{PM} = t_a + t_b \quad 6.58$$

which is equal to

$$T_{PM} = \frac{n}{p} (j - j' - 1) + n(2 - \frac{1}{p}) - (j' + 1) - \frac{1}{p} (2^j - 2P) \quad 6.59$$

After substituting j and j' by their values we obtain

$$T_{PM} = \frac{1}{p} \left[n \log \frac{M}{p} + 2n(P-1) + P(1 - \log P) - M \right] \quad 6.60$$

T_p , the run-time of the parallel Quicksort merge is given as the sum of 6.49 and 6.60

$$T_p = \frac{1}{p} \left[\frac{12}{7} (n+p) \left(H_{\frac{n}{m+1}} - H_{m+2} + 2M + (n+p) \right) \frac{37m-94}{49(m+2)} + n \log \frac{M}{p} \right]$$

$$+ 2n(P-1) + P(1 - \log P) - M] \quad 6.61$$

which is simplified as,

$$T_p = \frac{1}{P} \left[n \left\{ \frac{12}{7} \left(H_{\frac{n}{M}+1} - H_{m+2} \right) + \log \frac{M}{P} + \frac{37m-94}{49(m+2)} \right\} \right. \\ \left. + p \left\{ \frac{12}{7} \left(H_{\frac{n}{M}+1} - H_{m+2} \right) + P - \frac{12m-192}{49(m+2)} \right\} + M \right] \quad 6.62$$

Tables 6.9 and 6.10 report respectively the theoretical and experimental performance of the Parallel Quicksort merge algorithm.

M	P	T_p	S_p	E_p
8	2	5.27	1.29	.65
	3	4.53	1.50	.50
	4	3.60	1.89	.47
	5	3.62	1.88	.38
16	2	5.59	1.22	.61
	3	4.78	1.42	.47
	4	3.72	1.83	.46
	5	3.65	1.86	.37
32	2	5.92	1.15	.57
	3	4.91	1.38	.46
	4	3.92	1.73	.43
	5	3.77	1.80	.36
64	2	6.29	1.08	.54
	3	5.07	1.34	.45
	4	4.12	1.65	.41
	5	3.87	1.76	.35

TABLE 6.9: Experimental results of the PQM algorithm when $n = 16K$, $m = 10$ and n paths = $M \times P$

M	P	S_p	E_p	PF_p
8	2	1.786	.893	1.595
	3	2.504	.835	2.090
	4	2.953	.738	2.180
	5	3.489	.698	2.435
16	2	1.810	.905	1.637
	3	2.535	.845	2.142
	4	2.985	.746	2.227
	5	3.525	.705	2.485
32	2	1.834	.917	1.681
	3	2.566	.855	2.195
	4	3.017	.754	2.276
	5	3.562	.712	2.537
64	2	1.858	.929	1.726
	3	2.598	.866	2.250
	4	3.050	.763	2.326
	5	3.598	.720	2.589

TABLE 6.10: Theoretical results of the PQM when $M > P$. $n = 16K$ words and $m = 10$

From Table 6.9 we see that the efficiency slightly decreases as M increases. This is reflected also in the theoretical results as shown in Table 6.10. However, it is better to use the straight Parallel Quicksort since merging degrades considerably the increased performance gained during sorting.

6.3.3 PARALLEL PARTITIONED SORTING ALGORITHMS (PPS)

We observed in Section 6.2.4 that the parallel run-time of the PPS algorithms, T_p , is the sum of two run-times T_{PS} and T_{PW} . Let us first consider the run-time for the parallel partitioning and sorting phase T_{PS} . $\frac{M}{P}$ subsets, each containing $(\frac{n}{M})$ elements are sorted by every processor. Therefore T_{PS} is

$$T_{PS} = n \frac{M}{P} + \frac{M}{P} t_s \quad 6.63$$

where t_s is the run-time for the sequential Quicksort to sort a subset of length $(\frac{n}{M})$. The quantity $n \frac{M}{P}$ reflects the fact that every processor accesses every element of the set S_n exactly $\frac{M}{P}$ times when picking-up elements. T_{PS} is then

$$T_{PS} = n \frac{M}{P} + \frac{12}{7} \left(\frac{n}{M} + 1 \right) \left(H_{\frac{n}{M}+1} - H_{\frac{n}{M}+2} \right) + 2 \frac{M}{P} + \left(\frac{n}{M} + 1 \right) \frac{37m-94}{49(m+2)} \quad 6.64$$

If each of the processors writes $(\frac{M}{P})$ subsets to an auxiliary shared array before the final transfer to the original array, then,

$$T_{PW} = \frac{M}{P} \times \frac{n}{M} + \frac{n}{P} = 2 \frac{n}{P} \quad 6.65$$

The experimental runs of the 2 PPS show a very poor performance as M increases (see Tables 6.11 and 6.12). This is explained by the amount of synchronization and data transfer, since they are of order $(\frac{M}{P})$. As a suggestion to improve the performance of these PPS algorithms it is better to use for every path a local array to store and sort the M subsets. This only reduces the synchronization mechanism by half but not the data transfer. However, the 2 PPS performed very well when the task was split into exactly P subsets.

This improved performance was not achieved by the parallel Quicksort algorithm even for M as large as 256.

M	P	T_{pc}	T_{pe}	S_{pc}	S_{pe}	E_{pc}	E_{pe}
8	2	6.52	6.69	1.04	1.02	.52	.51
	4	4.71	4.76	1.44	1.43	.36	.36
16	2	8.13	8.22	.83	.83	.42	.41
	4	4.95	5.01	1.37	1.36	.34	.34
32	2	11.21	11.46	.60	.59	.30	.30
	4	6.06	6.17	1.12	1.10	.28	.28
64	2	18.67	19.00	.36	.36	.18	.18
	4	10.19	10.37	.67	.66	.17	.16

TABLE 6.11: Experimental results of the PBPS algorithm when $n = 16K$, $m = 10$ and $n \text{ paths} = M > P$

M	P	T_{pc}	T_{pe}	S_{pc}	S_{pe}	E_{pc}	E_{pe}
8	2	5.02	5.15	1.35	1.32	.68	.66
	4	2.58	2.72	2.63	2.50	.66	.63
16	2	6.57	6.73	1.03	1.01	.52	.51
	4	3.38	3.51	2.01	1.94	.50	.48
32	2	10.17	10.35	.67	.66	.33	.33
	4	5.20	5.27	1.30	1.29	.33	.32
64	2	17.54	17.90	.39	.38	.19	.19
	4	8.85	9.01	.77	.75	.19	.19

TABLE 6.12: Experimental results of the PRPS algorithm when $n = 16K$, $m = 10$ and n paths = $M \times P$

6.4 CONCLUSIONS

In this Chapter, we presented two new parallel sorting algorithms, the Bounded-Partitioned and Range-Partitioned Sorting algorithms. Unlike the Parallel Quicksort algorithm, the two new methods have the potential of partitioning in parallel the original set, S_n , into P independent subsets of nearly equal lengths and which can be sorted by P asynchronous processors. A fourth algorithm, the Parallel Quicksort-Merge was also considered but only as a comparable alternative to the parallel Quicksort algorithm.

The performance analysis which is based on counting the number of key comparisons was presented for all four parallel algorithms and for two different situations depending on whether or not the number of parallel paths (or subsets) exceeds the number of available processors P . The theoretical analysis showed, as anticipated, that the parallel Quicksort and the Parallel Quicksort-Merge have equal performance, whereas the Parallel Partitioned Sorting algorithms outperformed the Parallel Quicksort algorithm for $M=P$.

The validation of the theoretical analysis was supported by a series of experiments performed on the Sequent Balance 8000TM. A set of n randomly generated elements ($n = 16K$ words) was selected to be sorted by the four different algorithms. Two performance measures, S_p , the speed-up ratio and, E_p , the efficiency factor were selected as a means for comparing the performance of the Parallel Sorting algorithms.

The experimental results which are in close agreement with the theoretical results are also included in tabular or graphical form.

Chapter 7

A VLSI SOFT-SYSTOLIC IMPLEMENTATION OF A STRING PATTERN MATCHER AND ITS VARIANTS

7.1 INTRODUCTION TO THE VLSI TECHNOLOGY PARADIGM

Recently we have witnessed a rapid growth of computing technology that has followed the invention of transistors in the late 1940's. (The first transistor was invented in 1948 at the Bell Telephone Laboratories) and integrated circuits in the late 1960's. Through developments in transistors, new families of small computers (i.e. minicomputers) began to emerge on the market. As a result, thousands of transistor elements were assembled on minute chips of silicon. The race for smaller and faster computing machines has developed ever since. A mainframe computer built using the original thermionic valves had weighed more than thirty tons and required a room of 60 x 25 feet square to hold it; a computer of superior capability could, by 1971, be accommodated on a sliver of silicon.

The migration of IC to Large Scale Integration (LSI) technology allowed tens of thousands of electronic components to fit on a single chip. Following the rapid advances in LSI technology, the Very Large Scale Integration (VLSI) circuits have been developed with which enormously complex digital electronic systems can be fabricated on a single chip of silicon, one-tenth the size of a postage stamp. In fact, it is foreseen that the number of components that a VLSI chip could accommodate would be increased by a multiplier factor of ten to one hundred in the next two decades [Mead, 1980]. Devices which once required many complex components can now be built with just a few VLSI chips, reducing the difficulties in reliability, performance and heat dissipation that arise from standard SSI and MSI components [Kung, 1979].

As computer applications still require faster and more powerful computer architectures than those currently available and as we are migrating from the information processing era towards "knowledge" based systems which characterise the projected fifth generation of computers, the research in computer technology has been widened more than ever before. H.T. Kung was the first to realise that the rapidly developing chip industry together with automata theory could be the key success to constructing fast, highly parallel computer structures at low cost. Until the advent of VLSI, the development of parallel computers with a large number of processors had been limited by the unaffordable high costs of manufacture. Existing machines had been improved by tinkering with the traditional Von-Newmann architecture, for instance cycle stealing, direct memory access (DMA), and pipelining of fetch and execute operations. As such, parallel machines were confined only to research purposes or military operations.

The development of new manufacturing techniques for fabrication of small, dense and inexpensive semi-conductor chips created a unique circumstance in the computer industry. With the use of VLSI in circuits, size and cost of processing elements and memory was considerably reduced and it became feasible to combine the principles of automation theory with the pipeline concepts. The combination was especially attractive since device manufacture cost remained constant relative to circuit complexity, with most time and money invested in design and testing.

In relation with what was said above, approaches to device designs have progressed so significantly to the point that hardware design now relies heavily on software techniques, i.e. special rules for circuit layout and high level design languages (e.g. Geometry

languages, Stick languages, Register Transfer languages, etc) [Mead 1981]. In fact, some of these languages offer the powerful chip fabrication capability directly from a design they express.

Illustrative of this trend is the term **silicon compiler** utilised by hardware designers to refer to computer-aided design systems currently under development. Analogous to a conventional software compiler, the silicon compiler will convert linguistic representations of hardware components into machine code, which can be stored and subsequently utilized in computer-assisted fabrication.

The actual implementation of such designs requires a highly sophisticated manufacturing technology, found in **silicon wafer** fabrication. Such a technology exhibits the most powerful attribute which is its **pattern independency**. In other words, there is a clear distinction between the processing performed during wafer fabrication, and the design effort that creates the patterns to be implemented. This distinction requires a precise specification to the designer of the processing line capabilities. The specification usually takes the form of a set of permissible geometries that may be utilised by the designer with the knowledge that they are within the resolution of the process itself and that they do not violate the device physics required for the proper operation of transistors and interconnections formed by the process. When reduced to their simplest form, such geometrical restrictions are called **design rules**. These constraints are of the form of minimum allowable values for certain widths, separations, extensions and overlaps of geometrical objects, patterned in various system levels (see Mead and Conway [Mead 1980]).

Without going into any further details of the design rules, we must mention a characteristic and fundamental fact concerning the progressive miniaturisation of the minimum distance, within which one can expect what is deposited on the wafer actually to appear in the design of integrated circuits. This is that all dimensions in designs are specified not in absolute sizes, but in terms of multiples of an elementary distance parameter, the so called length-unit (λ). This parameter is, approximately, the maximum amount of 'accidental' displacement that we can expect when we deposit a feature on the wafer. In the early 1980s, λ was usually considered to be about 2 μm (i.e. micron).

Now if we try to sketch a complex automata arrangement one is immediately confined to the two dimensional (2D) plane defined by sheets of paper. In fact VLSI is achieved in a similar manner by a combination of circuit designs with high resolution photolithographic (or the newer X-ray photography) techniques, where it is convenient to place wires on rectangular grids, and limit the number of parallel layers of semi-conductors material containing wires and circuit elements. Hence, the problem of collapsing a three dimensional (3D) graph structure onto a 2D plane or chip, is simplified if the graph is as close to 2D as possible*. Furthermore, an 'almost' planar graph based circuit is easier to design if it is modular - i.e. composed of many replicatable components, and consequently reduces overall production time as only a single or a few cells must be designed.

However, VLSI presents some problems, as the size of wires and transistors approach the limits of photolithographic resolution, for

* A 2D graph is termed planar if it can be drawn on the plan with no axes intersecting at places other than nodes

it becomes literally impossible to achieve further miniaturisation and actual circuit area becomes a key issue. In addition, the chip area is also limited in order to maintain high chip yield and the number of pins (through which the chip communicates with the outside world) is limited by the finite size of the chip perimeter. These restrictions form the basis of the VLSI paradigm.

For a newly developed technology or product to survive in a highly competitive industry there must be sufficient demand for it. The emergence and subsequent success of VLSI oriented computing systems is not due only to H.T. Kung's foresight but also to the timing. At the same time Kung revealed the systolic concept, the idea of using VLSI for signal processing was the major focus of attention in governmental, industrial and university research establishments.

7.2 FUNDAMENTAL ARCHITECTURAL CONCEPTS IN DESIGNING SPECIAL PURPOSE VLSI COMPUTING STRUCTURES

High-performance special-purpose VLSI oriented computer systems are typically used to meet specific applications, or to off-load computations that are especially taxing to general-purpose computers. However since most of these systems are built on an ad hoc basis for specific tasks, methodological work in this area is rare. In an attempt to assist in correcting this ad hoc approach, some general design concepts will be discussed, while in the following paragraph the particular concept of systolic and wavefront array architectures, two general methodologies for mapping high-level computation problems into hardware cellular structures, will be introduced.

The problem of embedding a network of processors and memories into a set of VLSI chips is similar to that of embedding graphs, whose nodes are computers, or gates, onto grids so as to minimise area. Most of the researchers exploring this problem usually make certain assumptions; for example, they assume that wires run and devices are oriented in only horizontal and vertical directions, everything is embedded on a square grid, all device nodes are at the same layer.

The computational power of a chip is often measured by the number of transistors it contains. However, this is quite a misleading approach for the organisation of a chip's circuitry has a very strong effect. In general, regular chip designs make more efficient utilisation of silicon area, which is a more natural measurement factor for the circuit size than the number of transistors. Such designs utilise less area for the wiring amongst transistors, leaving more space for the transistors themselves.

From the memory capacity point of view, the number of bits has been quadrupling every few years; in the mid-1970s technology passed through the era of 1K, 4K and 16K bits memory chips. In 1981 the memory size was expanded to 32K bits and a 64K bit is predicted.

Particularly for the design of special-purpose VLSI oriented computer machines, cost effectiveness has always been a major concern; their fabrication must be low enough to justify their specialised, and consequently, limited applicability. Cost can be distinguished in non-recurring design and recurring part costs. Any fall of the latter's cost is equally applied for the merit of both special-purpose and general-purpose computer systems. Furthermore, this cost is even less significant than the design cost, since the

production of special-purpose computer systems in large quantities is quite a rare phenomenon. Hence, conclusively, the design of such a system should be relatively small for it to become more attractive compared to a general-purpose computer and this can be achieved by the utilisation of appropriate architectures. More specifically, if the decomposition of a structure into a few types of simple sub-structures which are repetitively utilised with simple and regular interfaces is feasible, then significant savings are most likely to be achieved.

In addition, special-purpose computer systems based on simple and regular designs are likely to be modular and consequently adjustable to various performance goals - i.e. system costs may be made analogous to the performance required. This fact reveals that achieving the architectural challenge for simple and regular design, yields cost-effective special-purpose computer systems.

Since such VLSI computing structures can function as peripheral devices, attached to a conventional host computer, receiving data and control signals and outputting results, a computation rate, which will balance the available I/O bandwidth with the host, is the ultimate performance goal of a special-purpose computer system. Therefore the likely modular attribute of such a concept is highly necessary, since it allows the flexibility of the structure to match a variety of I/O bandwidths; and since an accurate a priori estimate of available I/O bandwidths in complex systems is often possible.

However this problem becomes especially severe when a very large computation is performed on a relatively small special-purpose computer system. In this case the computation must be decomposed.

In fact one of the major challenging research items becomes the development of algorithms that could be mapped into and executed efficiently by a special-purpose computer system. This implies that algorithms should decompose into modules, that map compactly into one VLSI chip (or a module of chips), and modules should be interconnected in an efficient manner. These algorithms must support high degrees of concurrency and employ a simple, regular data and control flow to enable an efficient implementation.

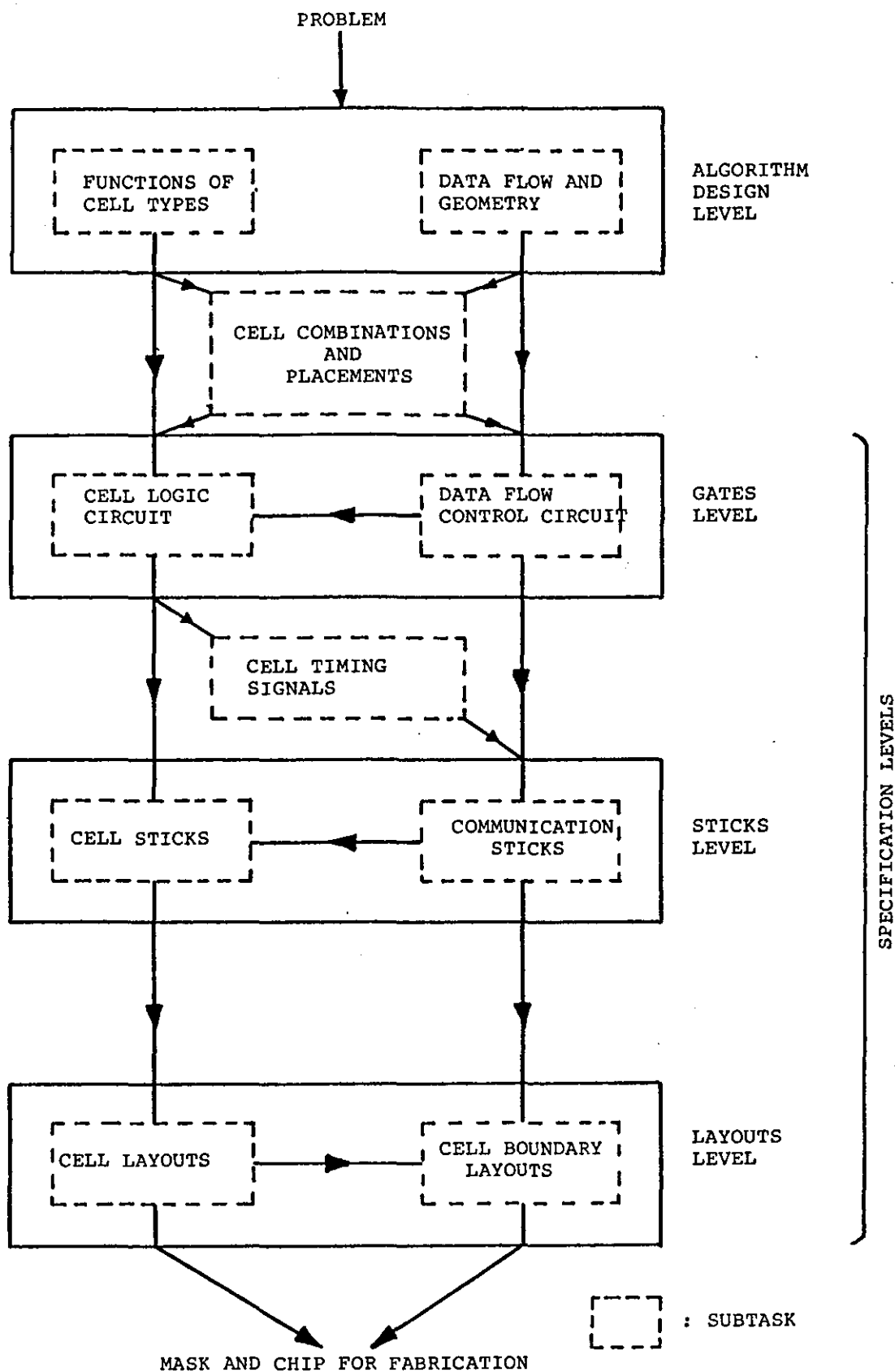
To conclude we mention that special-purpose VLSI oriented computing structures can be either a single chip, built from a replication of simple cells, or a system built from identical chips, or even a combination of these two approaches. Figure 7.1 summarises the principle stages and tasks interdependencies involved in the design of a VSLI chip (see Foster and Kung's paper, [Foster 1980]). In fact in the environment of VLSI systems design, the boundary between software and hardware has become increasingly vague.

7.2.1 SYSTOLIC ARRAYS

The concept of systolic architectures, pioneered by H.T. Kung, which has been successfully shown to be suitable for VLSI implementation is basically a general methodology of directly mapping algorithms onto an array of processor elements. It is especially amenable to a special class of algorithms, taking advantage of their regular, localised data flow.

The word 'systole' was borrowed from physiologists who used it to describe the rhythmically recurrent contraction of the heart and arteries which pulse blood through the body. By analogy, the function of a cell in a systolic computing system is to ensure that

FIGURE 7.1: THE DESIGN STAGES OF A SPECIAL-PURPOSE VLSI CHIP



data and control are pumped in and out to a regular pulse, while performing some short computation [Kung 1978].

A systolic array is a network of processing elements, usually arranged in a regular pattern and locally linked by communication channels. Operands are pumped through the array to a regular pulse. Everything is planned in advance so that all inputs to a cell arrive at just the right time before they are consumed. Intermediate results are passed on immediately to become the inputs for further cells. A steady stream flows at one end of the array which is said to consume data and produce results on the 'fly'. For instance, by locally connecting a few basic cells, almost known as Inner Product Steps - 'IPS' - each performing the operation $C = C + A \times B$ - leads to a fundamental network capable of performing computation-intensive algorithms, such as digital filtering, matrix multiplication, and other related problems (see Table 4.1 for a more comprehensive list of potential systolic applications).

The systolic array systems feature the important properties of modularity, regularity, local interconnection, a high degree of pipelining and highly synchronised multiprocessing. Such features are particularly more interesting in the implementation of compute-bound algorithms, rather than Input/Output - 'I/O' - bound computations. In a compute-bound algorithm, the number of computing operations is larger than the total number of I/O elements, otherwise the problem is termed I/O-bound. Illustrative of these concepts are the following matrix-matrix multiplication and addition examples. An ordinary algorithm, for the former, represents a compute-bound task, since every entry in the matrix is multiplied by all the entries in some row or column of the other matrix - i.e. $O(n^3)$ multiply-add steps, but only $O(n^2)$ I/O elements. The addition

<u>'SYSTOLIC' PROCESSOR</u>	<u>PROBLEM CASES</u>
<u>ARRAY STRUCTURE</u>	
<i>1-D linear arrays</i>	: FIR-filter, convolution, 'Discrete Fourier Transform' - <i>DFT</i> , matrix-vector multiplication, recurrence evaluation, solution of triangular linear systems, carry pipelining, Cartesian product, odd-even transposition sort, real-time priority queue, pipeline arithmetic units.
<i>2-D square arrays</i>	: Dynamic programming for optimal parenthesization, image processing, pattern matching, numerical relaxation, graph algorithms involving adjacency matrices.
<i>2-D hexagonal arrays</i>	: Matrix problems (matrix multiplication, <i>LU</i> -decomposition by Gaussian elimination without pivoting, <i>QR</i> -factorization), transitive closure, relational database operations, <i>DFT</i> .
<i>Trees</i>	: Searching algorithms (queries on nearest neighbour, rank, etc., systolic search tree), recurrence evaluation.
<i>Triangular arrays</i>	: Inversion of triangular matrix, formal language recognition.

TABLE 7.1: THE POTENTIAL UTILIZATION OF 'SYSTOLIC' ARRAY CONFIGURATIONS

of two matrices, on the other hand, is an I/O bound task, since the total number of adds is not larger than the total number of I/O operations - i.e. $O(n^2)$ add steps and $O(n^2)$ I/O elements.

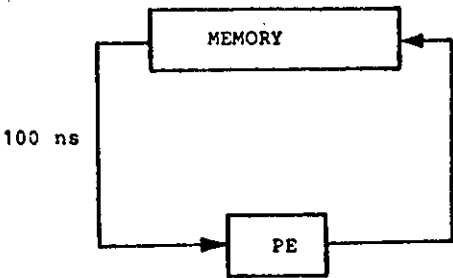
It is apparent that any attempt to speed-up an I/O-bound computation must rely on an increase in memory bandwidth (the so-called 'Von Neumann' bottlenecks). Memory bandwidths can be increased by the utilisation of either fast components, which may be quite expensive, or interleaved memories, which may create complex memory management problems. However, the speed-up of a compute-bound computation may often be achieved in a relatively simple and inexpensive manner, that is by the systolic architectural approach.

The fundamental principle of a systolic architecture, a systolic array in particular, is illustrated in Figure 7.2. By replacing a single processing element with an array of PEs, a higher computation throughput can be achieved without increasing memory bandwidth. This is apparent if we assume that the clock period of each PE is 100 ns; then the conventional memory-processor organisation (a) has at most 5 MOPS performance, while with the same clock rate, the systolic array (b) will result in a possible 35 MOPS performance.

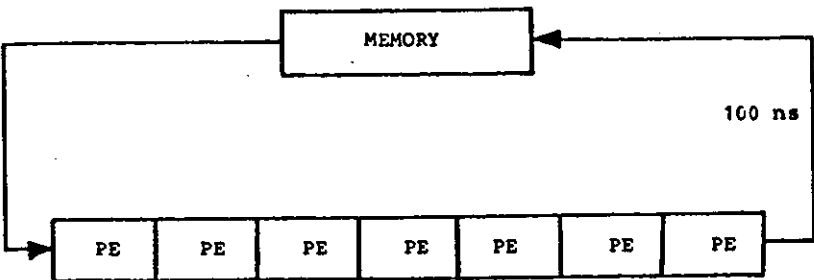
Finally this approach of utilising each input data item a number of times, thus achieving a high computation throughput with only a modest memory bandwidth, is just one of the advantages of the systolic concept. Other equally significant criteria and advantages include modular expansibility, utilisation of simple, uniform cells, extensive concurrency and fast response time.

However, one problem associated with systolic array systems, is that the data and control movements are controlled by global timing-

reference beats. In order to synchronise the cells, extra delays are often used to ensure correct timing. More critically, the burden of having to synchronise the entire network will eventually become intolerable for very large or ultra large scale arrays.



a) The Conventional Organization



b) A Systolic Processor Array

FIGURE 7.2: SYSTOLIC DESIGN PRINCIPLE

7.2.2 WAVEFRONT ARRAYS

A solution to the above mentioned problems, as suggested by S.Y. Kung [Kung 1985], is to take advantage of the data and control flow locality, inherently possessed by most algorithms. This permits a

data-driven, self-timed approach to array processing. Conceptually, such an approach substitutes the requirement of correct 'timing' by correct 'sequencing'. This concept is used extensively in data flow computers and wavefront arrays.

Basically the derivation of a wavefront process consists of the three following steps:

- a) the algorithms are expressed in terms of a sequence of recursions;
- b) each of the above recursions is mapped to a corresponding computation wavefront; and
- c) the wavefronts are successively pipelined through the processor array.

Based on this approach, S.Y. Kung introduced the Wavefront Array Processor (WAP) which consists of an $N \times N$ processing element with regular connection structure, a program store and memory buffering modules as illustrated in Figure 7.3. The processor grid acts as a wave propagating medium using handshaking protocols.

Each processor performs a limited number of computations and is controlled by a program loaded in the program store. Data is stored in memory modules around the boundary and extra time must be allowed to set up a computation. An algorithm is executed by a series of wavefronts moving across the grid with processors computing whenever its data and instructions are available. Processors are assumed to support pipelining of waves and the spacing of waves (T) is determined by the availability of data and the execution of the basic operation. The speed of the wavefront Δ is equivalent to the data transfer time.

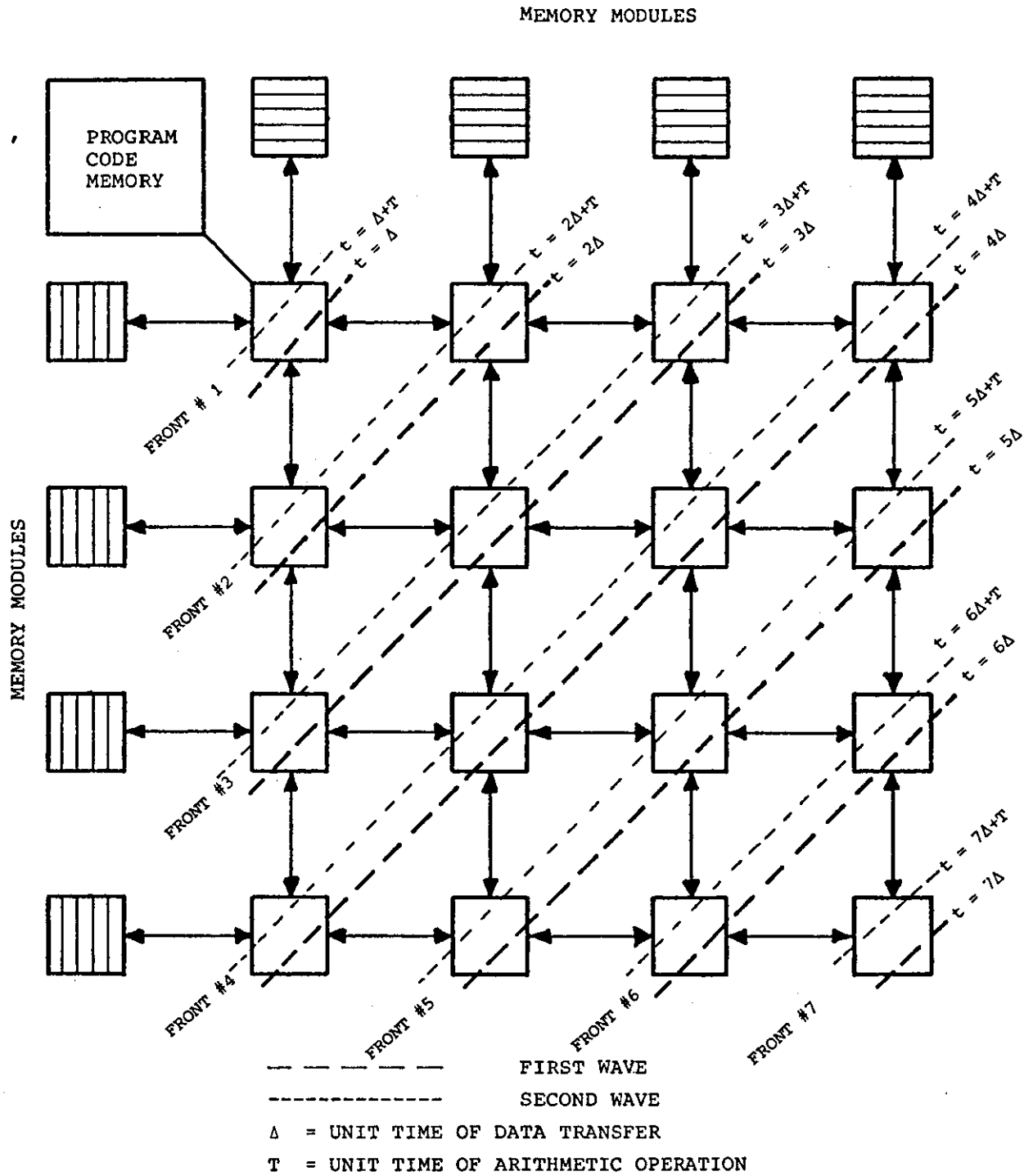


FIGURE 7.3: THE WAVEFRONT ARRAY PROCESSOR

Summarising, the wavefront approach combines the advantages of data flow machines with both the localities of data flow and control flow inherent in a certain class of algorithms. Since the burden of synchronising the entire array is avoided, a wavefront array is architecturally 'scalable'.

7.3 VLSI-ORIENTED ARCHITECTURES

For large applications it may not be feasible to design a single chip implementation of an array, especially when balance between flexibility, efficiency, performance and implementation cost is essential. An alternative approach is to implement basic cells at the board level using a set of 'off-the-shelf' components which are widely available as chip packages from various manufacturers.

Systolic arrays achieve high performance and efficiency by considering only restricted problem classes, at the expense of flexibility and implementation cost. For a more economical solution, arrays must be constructed with many incorporated features so as to handle a large number of systolic algorithms. In this section, we shall briefly review the main contenders of VLSI-oriented computing systems which have received attention to date.

7.3.1 THE WARP ARCHITECTURE

The WARP architecture, one of the most advanced VLSI-oriented systems, was developed at Carnegie Mellon University (CMU) by H.T. Kung and his associates for purely systolic algorithms. Initially, the design began with a preliminary study of different architectures based on general purpose microprocessors which could implement a variety of systolic algorithms efficiently. The study resulted in

the Programmable Systolic Chip (PSC) discussed in [Fisher 1984] and prompted research into cell structures for high performance systolic arrays in a particular area (signal processing).

The WARP architecture is a 1D linear systolic array with data and control flowing in one direction (with input at one end of the array and output at the other). From the preceding discussions we observe that the design allows easy implementation, synchronization by a simple global clock mechanism, minimum input/output requirements and the use of efficient fault tolerance techniques for faults.

The basic WARP cell is constructed from a collection of chips as is illustrated in Figure 7.4, its main characteristics being the pipelining of data and control. Weitek 32-bit floating point multiplier (MPY) and ALU perform operations and can be used in pipeline mode to improve throughput by two level pipelining. The MPY and ALU register files use Weitek register file chips and can compute approximate functions like inverse square root using look-up facilities. The *x,y* and *addr*-files are also register files but this time used to implement delays for synchronising data paths, and can be used as extra registers for book-keeping operations, while the data memory is used to reduce the input/output bandwidth by implementing tables of data and storing intermediate results, it can also be used to implement multiple cells on the same processor and hence 2D arrays. The crossbar and input multiplexors (muxes) provide communication between the individual elements and can be reconfigured by control signals. The muxes permit two-directional data flow and ring set-ups. A ten-cell prototype has been built at CMU and tested on a number of example arrays discussed in H.T. Kung, [Kung 1984a].

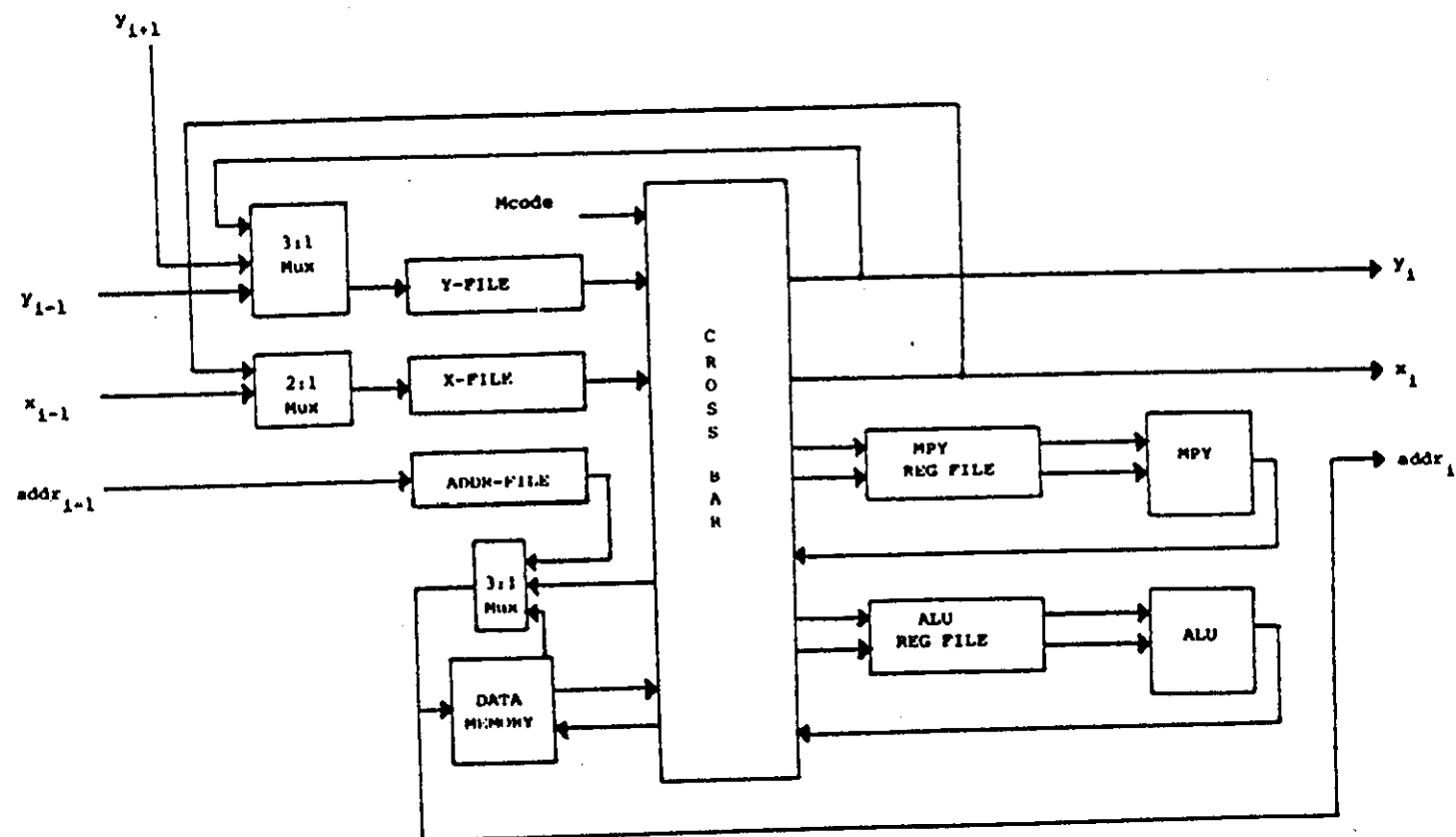


FIGURE 7.4: DATA PATHS FOR THE WARP CELL

7.3.2 THE CHIP ARCHITECTURE

In order to derive a more flexible VLSI-oriented computing system than the special-purpose computers, where the same hardware would be used to solve several different problems, L. Snyder suggested the design of the Configurable, Highly Parallel architecture - 'CHiP' [Snyder, 1982] based on the configurability principle. Conceptually, the CHiP represents a family of systems, each built out of three major components: a set of processing elements (PEs), a switch lattice and a controller.

The lattice, the most important component of a CHiP, is a 2D structure of programmable switches connected by data paths. PEs are placed at regular intervals. Figure 7.5 shows two examples where squares represent PEs, circles represent switches and lines represent data paths. Note that PEs are not directly connected to each other, but rather are connected to switches.

The processing elements are microprocessors each coupled with several kilo-bytes of RAM used as local storage. Data can be read or written through any of the eight data paths or ports connected to the PE. Generally, the data transfer unit is a word, though the physical data path may be narrower. The PEs operate synchronously and systolically.

Each programmable switch contains a small amount (around 16 words) of local RAM which is used to store instructions (one instruction per every word) called configuration settings. Each configuration setting specifies pairs of data paths to be connected. When executed, each pair which is also known as a crossover level, establishes a direct, static connection across the switch that is

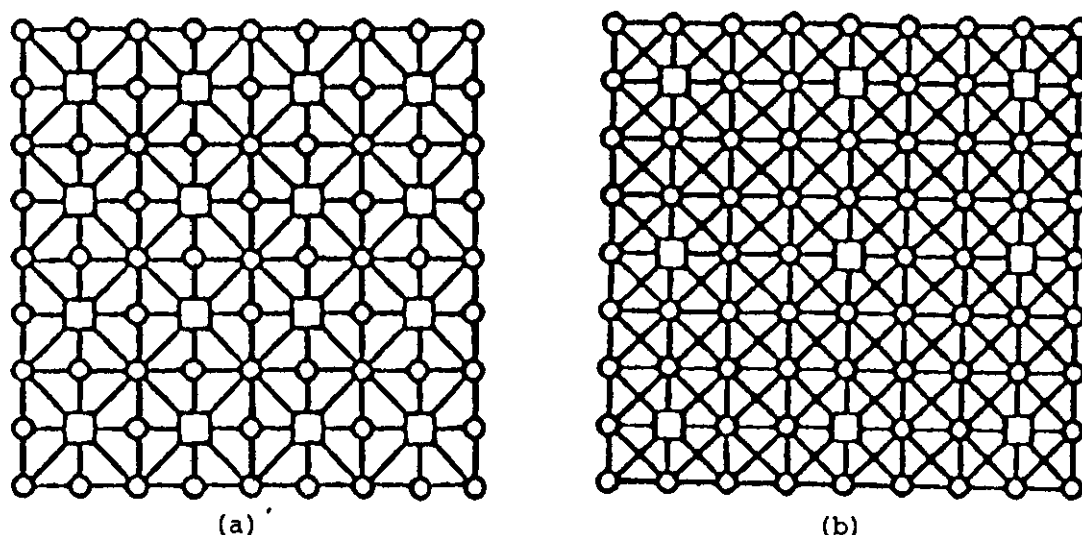
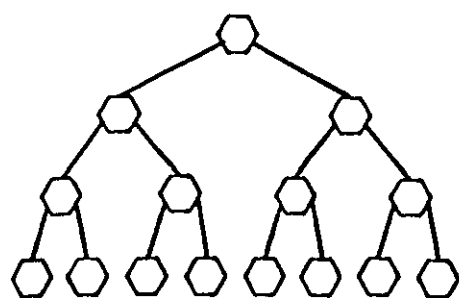


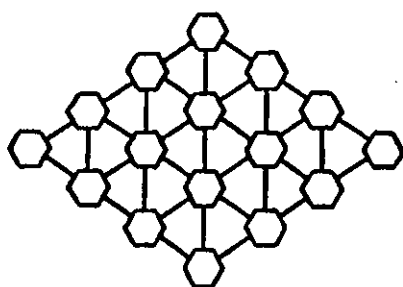
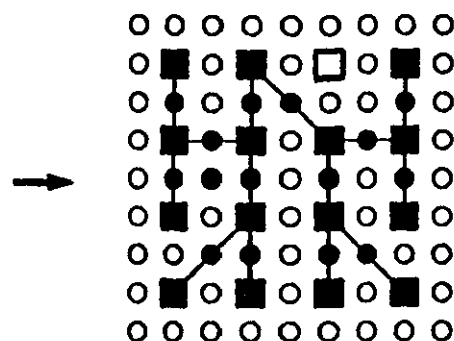
FIGURE 7.5: TWO LATTICE STRUCTURES.

independent of the others. The data paths are bidirectional and fully duplex, i.e. data movements can take place in either direction simultaneously. Now, executing a configuration settings program causes the specified connections to be established and to persist over time, e.g. over the execution of an entire algorithm.

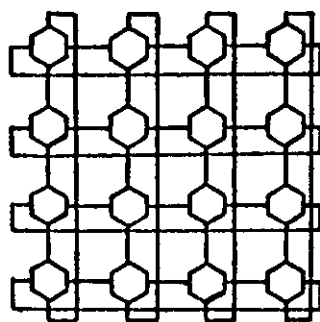
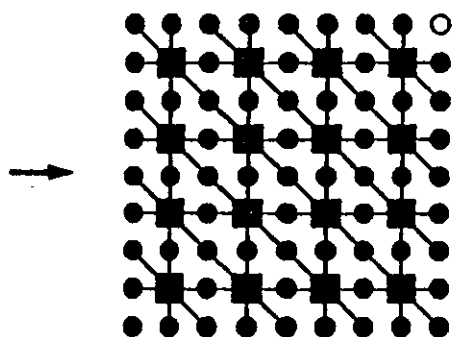
The processing elements can be connected together to form a particular structure by directly configuring the lattice. That is, the programmer sets each switch such that collectively they implement the desired processor interconnection graph. Figure 7.6 illustrates three examples of how the lattice of Figure 7.5(a) might be configured to implement some commonly used interconnection schemes.



(a) Binary tree



(b) Systolic array



(c) Four-neighbour network

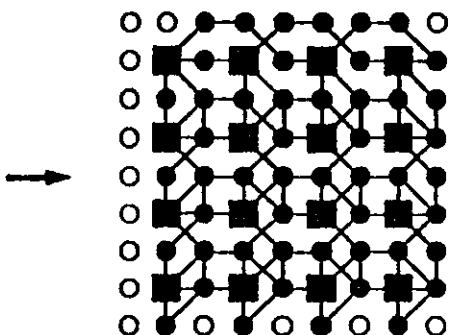


FIGURE 7.6: EMBEDDING GRAPHS INTO THE LATTICE OF FIGURE 7.5

In addition to the lattice, a controller is also provided, and is responsible for loading programs and configuration settings into PE and switch memories respectively. This task is performed through an additional data path network, called 'skeleton'.

From the functional point of view, CHiP processing starts with the controller broadcasting a command to all switches to invoke a particular configuration setting; for example to implement a mesh pattern. The established configuration remains during the execution of a particular phase of an algorithm. When a new phase of processing, requiring different configuration settings, is to begin, the controller broadcasts a command to all switches so that they invoke the new configuration setting; for example, a structure implementing a tree. With the lattice thus restructured, the PEs resume processing, having taken only a single logical step in reconfiguring the structure.

In conclusion, the CHiP computer which is a highly parallel computing system, providing a programmable interconnection structure integrated with the processor elements, is well suited for VLSI implementation. Its main objective is to provide the flexibility needed in order to solve general problems while retaining the benefits of regularity and locality.

7.3.3 INMOS TRANSPUTERS AND OCCAM

A third possibility is the INMOS transputer, a single chip microprocessor containing a memory, processor and communication links for connection to other transputers, which provides direct

hardware support for the parallel language OCCAM*. The structure of a transputer is given in Figure 7.7.

The transputer and OCCAM were designed in conjunction and all transputers include special instructions and hardware which provide optimal implementations of the OCCAM model of concurrency and communication. Different types of transputers can have different instruction sets depending on the required balance between cost, performance, internal concurrency and hardware, without altering the users view of OCCAM. Hence the transputer is a Reduced Instruction Set Computer (RISC).

The processor contains a scheduler which enables any number of processes to run on a single transputer sharing processor time, while each link provides two unidirectional channels for point to point communication synchronised by a handshaking protocol. Communication on any link can occur concurrently with communication on other links and with program execution.

OCCAM itself is based on communicating sequential processors [Hoare 1978] where parallel activities are viewed as black boxes with internal states, called processes, and which communicate with each other using a one-way channel. Communication is achieved by sending a message down a channel between two processes; one process sends a message and the other reads it from the channel.

* This language is named after the medieval philosopher who pioneered the idea of Occam's razor, a sharp intellectual instrument used to cut away all superfluous details in a system.

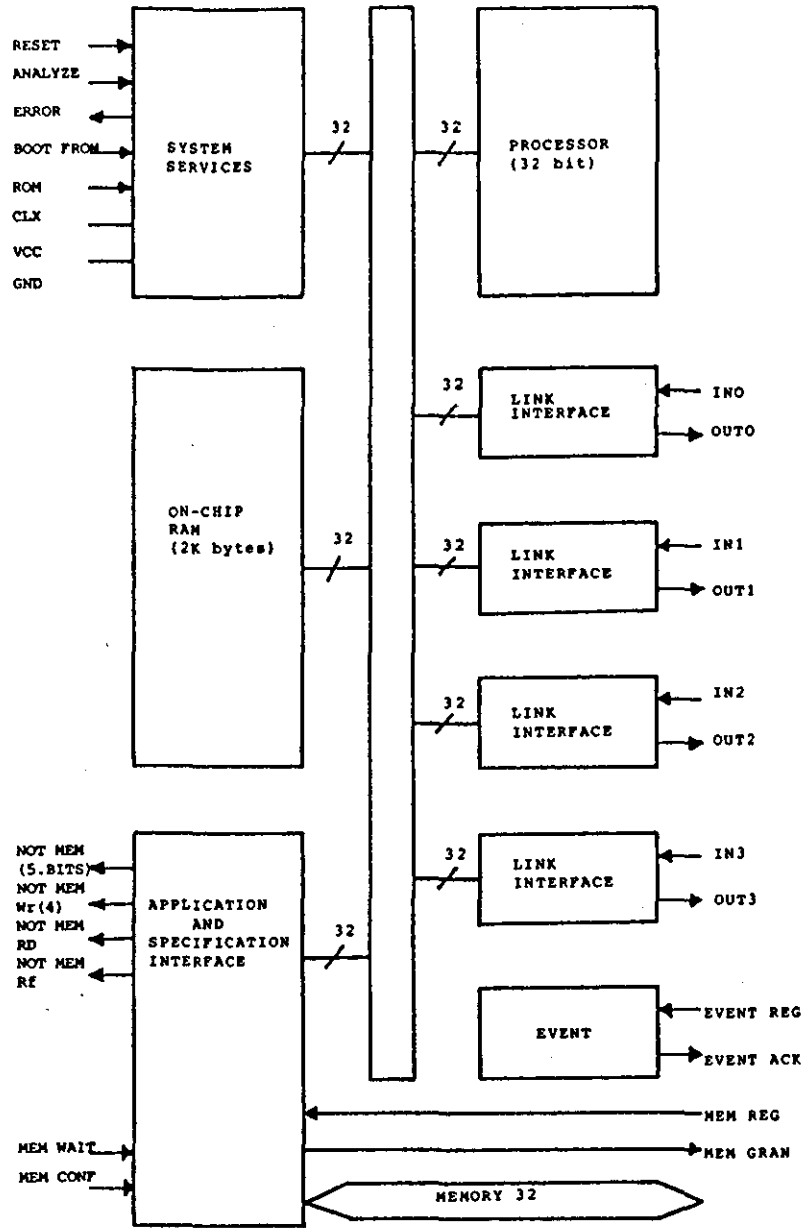


FIGURE 7.7: TRANSPUTER ARCHITECTURE

As every transputer implements OCCAM, an OCCAM program can be executed on a single transputer or a network of transputers. In the former case, parallel processes share the processor time and channel communication is simulated by moving data in memory. For a transputer network processes are distributed among transputers and channels allocated to links.

The main characteristic of the OCCAM language is its simplicity which makes it an appealing prospect for proving the correctness of processes. It has fewer than thirty keywords, and only a small number of constructors. Although each process uses destructive assignments, the use of channels for interprocess communication makes it entirely consistent with data flow and graph reduction computer architectures. OCCAM was designed with computer architectures of this nature in mind, and with a view towards fifth generation applications. Together with the Inmos transputers, it provides a modular hardware/software component of the type which is essential in the construction of highly parallel computer systems. However, its lack of a powerful data structure and its closeness to the hardware, means that OCCAM is likely to be the low-level language of fifth generation systems with applications possibly written in a more abstract language.

7.3.4 SIMULATION OF SYSTOLIC ARRAYS

We use the fact that OCCAM programs can be divorced from transputer configurations by using the language as a simulation tool throughout the remainder of this chapter for testing many proposed designs. A brief summary of the OCCAM language is given in Appendix B, together with selected simulated systolic programs. Figure 7.8 indicates the general structure of the programs, where branching indicates parallel execution. The construction of programs follows ideas developed by M.G. Megson [Megson 1984]. Consequently OCCAM programs simulate the formal proofs by replacing I/O descriptions by actual results. Although the simulation does not guarantee correctness it is nevertheless a less time consuming approach which does not result in unsolvable equations. Furthermore, a working OCCAM program

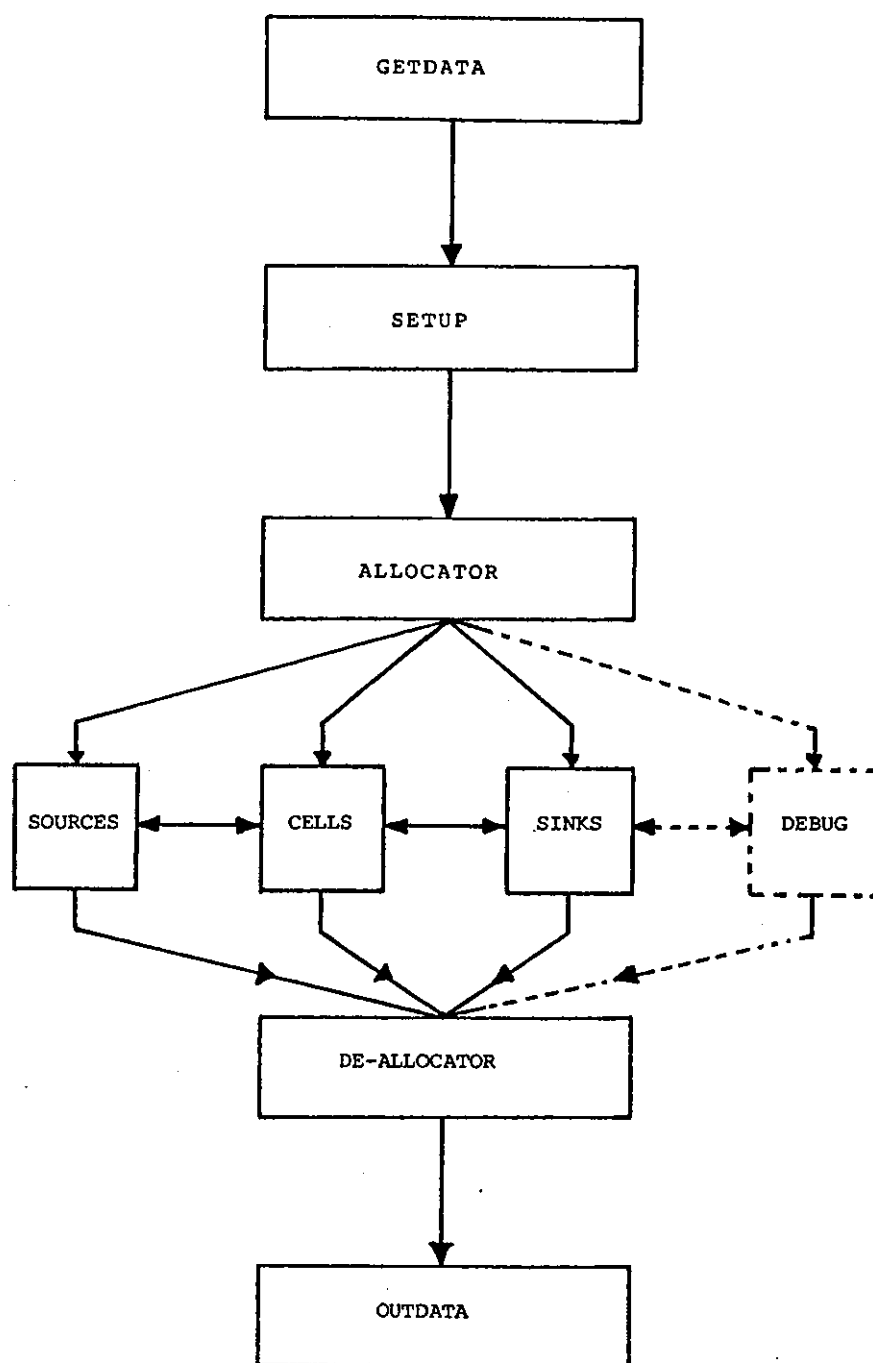


FIGURE 7.8: STRUCTURE OF OCCAM PROGRAM FOR SIMULATING SYSTOLIC ARRAYS

retains the possibility of actual transputer implementation and so solves two problems in one attempt.

The `getdata` and `putdata` sections of Figure 7.8 which represent the host machine interface, are responsible for receiving and sending data and control to and from the program. Each routine contains enough memory to store the initial array input data and the final output data corresponding to the global input and output sequences of the model. In principle, the two routines can be run in parallel with each other and the array, but generally they are sequential, in order to emphasise the parallel operation of the array. The actual host can be predefined I/O files or simply the terminal. The former method is useful for buffering and throughput testing, while the latter helps with debugging and interactive array performance. The routines can be augmented with user friendly features directing the program use, the collection of data necessary for the array construction and formatting of results.

The `setup` routine is a key section of the algorithm which computes array dependent quantities. More specifically, it performs many necessary calculations whose values are useful in defining the structure of the array. These structural values are more important as the array becomes more complex.

Sources, sinks and cells are OCCAM procedures that define the network model. A source is loaded initially with a vector from `getdata` representing its associated bounded data sequence, together with additional values from the set up routine. Sinks are analogous to sources except they work in reverse by placing real values into data vectors which are then passed to `putdata` for output. The cell procedures implement the n-ary sequence operators. Generally there

is one procedure for each type of cell, and the programming task is simplified for homogeneous networks. The I/O sequences are represented by OCCAM channels appearing as actual parameters in the procedure headings. Where cell definitions are only marginally different, extra switches and flags can be added to a procedure heading so it can set up the correct cell type. This collapses a number of definitions onto a single generic one. Extra parameters can also be used for preloading array values.

A cell definition is divided into three sections, **initialization**, **communication** and **computation**. Initialization is performed only once and allows cells to be cleared before use or predetermined values to be set up. In particular, initialization defines neutral element quantities which can be used in communication before real data reaches the cell, and is essential to maintain dataflow in OCCAM programs. The communication and computation sections of the cell are performed many times and are enclosed in a loop for iteration, and are performed sequentially one after the other. All communication is performed in parallel and computation is mainly sequential.

The **Allocator** routine is called after setup and is supplied with parameters about the array dimensions, synchronisation details and the total number of cycles in the algorithm if a loop scheme is used, and data sequence sizes. The allocator is simply a set of parallel loops which specify and start-up the computational graph by connecting corresponding procedures using OCCAM channels as arcs and allocating channels accordingly. To achieve setup, the graph is mapped onto a grid of points whose points and hence arcs can be recovered from a simple address type calculation. The simpler the array the easier are the mapping functions, and the result is an

allocation similar to the VLSI grid model. Once started the sources and sinks control computation, and the allocator only terminates when all the graph cell procedures have terminated. Termination of procedures is assumed to be globally synchronised if a for-loop is used in cells and asynchronous if while-loops are incorporated. As OCCAM is an asynchronous communication language, for-loops tend to be messy requiring some additional computation after the loop to clear all the channels - hence avoiding deadlock. While-loops are better suited to the model of concurrency and when augmented with systolic control sequences can be used to selectively close down cells input and output channels. Consequently array cells can be switched off or deallocated by a wavefront progression or pipelined approach from sources to sinks.

An additional procedure for debugging purposes can be added which runs in parallel with graph networks, and is mainly a screen/file mixer routine. The allocator sets up the procedure and network cells are augmented with an additional channel each, which the debug routine uses to analyse cells. Debug channels are allocated from a pool of channels and require an ordering of network cells for correct indexing. When the indexing function is simple, debug can be used to output snapshots of array operation so data flow can be easily verified. Snapshots are output in a sequential cell-ordering and the additional debug channel communication must be placed carefully in cell definitions.

Finally, the techniques described above have been used successfully throughout this chapter to implement designs in OCCAM but can in principle be extended to any parallel language provided channels and cells can be modelled. In fact Brent, Kung and Luk [Brent 1983] used an extended version of Pascal, ADA also seems a likely

candidate as ADA rendezvous is very similar to channel communication both being based on CSP. We adopt OCCAM because it offers more direct hardware support for special purpose designs as well as common architectures.

7.4 SYSTOLIC ALGORITHMS, CONSTRAINTS AND CLASSIFICATION

An algorithm that is designed with the systolic concepts in mind, in particular the use of simple and regular data and control flow, extensive use of pipelining and high level of multiprocessing, is termed a systolic algorithm. Technologically speaking, the design of systolic algorithms is in its early days, and as such, is applicable to only a small subset of applications. However, it is forecasted that further developments in the near future could alleviate some (if not all) of the restrictive constraints of the VLSI design.

Recent developments in programming languages along with the chip technology has made it possible to classify systolic algorithms into broad classes dependent on their specific properties. For example, a systolic algorithm can be considered upon many factors, i.e. ease of manufacture, its ability to be represented as a planar graph, or the amount of area required on silicon to implement it. Two main classes of systolic algorithms were identified [Bekakos 1986]: Hard-systolic algorithms and soft-systolic algorithms.

The hard-systolic algorithms represent the traditional algorithms designed with the physical chip implementation restrictions in mind so that they are easily manufactured as chip systems, examples include banded matrix-vector and matrix-matrix multiplication chips [Mead 1980].

Perhaps one of the most significant constraints imposed on VLSI systems is that it is a 2D technology (planarity constraint) since chips are usually (or more precisely wafered, if fabrication jargon is used) on a board. This physical constraint is reflected on the hard-systolic design by considering only those graph model representations which feature the planarity characteristic. However near planar representations are also allowed since the 2D constraint is violated by permitting two boards to be connected at some places.

In addition, broadcasting has been avoided in such algorithms since each cell has to be connected to the broadcast channel, increasing the power requirement of the system as a whole or decreasing its speed. In a 'purely' hard-systolic algorithm, broadcasting to cells is totally avoided. However, if only a limited amount* is allowed the algorithm is termed 'semi' hard-systolic algorithm.

The above constraints imposed on the hard-systolic algorithms are found to be very rigid and very closely related to the actual state of the VLSI technology and to fabrication problems. Although they were arguably shown to be mandatory conditions for a successful production of an efficient hard solution, however, they unnecessarily limit the inherent potential of the systolic approach (see the systolic programming paradigm [Shapiro 1984]).

A more flexible class of algorithms, the soft-systolic algorithms, were defined as a result of the innovations in the concurrent programming languages, such as OCCAM and CONCURRENT PROLOG. In such a class, planarity, broadcasting and area are no longer a major

* Bearing in mind that broadcasting over long distances could develop clock skews and that data cannot be synchronised

concern. Although the soft-systolic algorithms may intuitively not be suitable for direct mapping onto a chip, they however can still be performed on some suitable parallel computers, such as transputers. Therefore, these algorithms must be implemented in some appropriate languages.

Recent developments in the transputer device, in particular, the inclusion of a stored OCCAM compiler as a chip, have made the transputer chip a favourable candidate system to run some algorithms of this second class.

Evidently it is clear that the set of hard-systolic algorithms form a sub-set of the soft-systolic class and as such they can also be implemented in the same concurrent programming languages, although this is not necessary. Furthermore, it is also evident that some of the soft-systolic algorithms will be very close to the hard-systolic ones but, under the strict definitions of hard-systolic, would not be classed as such. Consequently, a third class, **hybrid-systolic** algorithms, was defined to represent this state of transition from the soft class to the hard one. Only technological improvements which are likely to take place in the near future will achieve this hybrid-hard migration. Current research indicates that algorithms which allow local broadcasting (not necessarily between nearest-neighbour cells), limited non-planarity or large amounts of non-planarity (but in a controlled manner) could be considered as contenders for this class of algorithm.

It is foreseen that all the above definitions will become increasingly important as the fifth generation of computer systems evolves. The relationship between these classes of algorithms, in a set theory manner, are given below:

1. $H_S \cup S_S = S$, and
2. $H_S \cap H_S \cap S_S$

where H_S, S_S and H_S are set symbols used to represent the hard-, soft- and hybrid-systolic algorithms respectively. It would be interestingly important to determine whether $H_S = S_S$ because, if this is the case, then all soft-systolic algorithms can, in principle, be fabricated. In the following section, the systolic principles will be demonstrated in various systolic designs when we study a family of soft-systolic pattern matcher algorithms, in particular when broadcasting to cells, limited cell storage and fan-in properties are considered.

7.5 SYSTOLISATION OF THE PATTERN MACHINING PROBLEM AND ITS VARIANTS

The importance of the string matching problem is well recognised in most computer applications. String pattern matching is a basic operation in SNOBOL-like languages and database query languages. Many artificial intelligence systems make substantial use of the string matching strategy as a search method. In general, searching is a very important topic in artificial intelligence and is currently under intensive study. Therefore, the design of an efficient pattern matcher chip could be very beneficial in both time and space savings for many computer applications. Furthermore, string pattern matching is similar to many stressing numerical computations such as convolutions and correlations. Though the above list is not complete (and was not intended to be so), its purpose is only to show some string pattern matching applications and, more importantly, to stress its importance as a general computer topic.

The first ever "purely" hard-systolic algorithm for the string pattern matching problem was developed at Carnegie Mellon University in 1979 by M.J. Foster and H.T. Kung [Foster 1980] who were the first to introduce the concept of systolic arrays. The design of the underlying algorithm demonstrated successfully the great potential of the then proposed special-purpose VLSI-oriented chip design methodology which is basically placed on the selection of a "good" algorithm. A "good" algorithm, in this context, should exhibit the following properties. The implementation of the algorithm should require a limited number of different types of simple cells, data flow and control flow in the network should be simple and regular and thirdly the algorithm should use extensive pipelining and multiprocessing. Accordingly, it was shown that such a "good" algorithm could be mapped onto circuits and layouts design for chip manufacturing in a most straightforward way.

We have already seen several known fast algorithms for the pattern matching problem that run on a Von-Newmann type computer system and developed parallel versions for them to run on any MIMD type asynchronous parallel computer system [Ghanemi, 1986a]. These algorithms (whether sequential or parallel) use a preprocessed table of information about partial matches of the pattern against itself. The purpose of this practice which, although consumes a fraction of the total amount of the execution time, is to avoid redundant comparisons, skipping over parts of the string where partial match results may be inferred from previous comparisons. Although the Boyer-Moore fast pattern matching method achieves a sub-linear performance, the systolic implementation of the brute-force algorithm greatly improves the throughput. This implies that time spent in I/O, control and data movements, as well as arithmetic operations are thus greatly reduced. Consequently, the Foster-Kung

pattern matcher chip solves the problem in almost linear time by comparing characters in parallel*.

A brief description of the Foster-Kung pattern matcher algorithm is presented in the following section. This would enable us to first assess the strengths and drawbacks of the design and then to be able to compare it with many of the soft-systolic designs which shall be presented in later sections.

7.5.1 HARD-SYSTOLIC DESIGNS

In this section, we shall review the hard-systolic pattern matcher chip as developed by M.J. Foster and H.T. Kung [Foster 1980]. We shall only concern ourselves with the design of the algorithm since fabrication techniques enable automatic mapping of the systolic algorithms onto circuits and layout designs for chip manufacturing. The systolic algorithm for the pattern matching problem is best presented by describing the data and control flows and the functions performed by its basic cell.

In the hard-systolic array, design R_1 , as proposed by Foster and Kung, the pattern and text string characters, denoted by $P_1 P_2 \dots P_k$ (k being the length of the pattern) and $S_1 S_2 \dots S_n$ respectively, move systolically in opposite directions through the array of cells. They alternatively arrive over the bus one character at a time, known as a beat or cycle. Thus, during each pair of consecutive beats the array inputs two characters and outputs one match result (A bit).

* Backing up the string in the case of a mismatched character is thus avoided

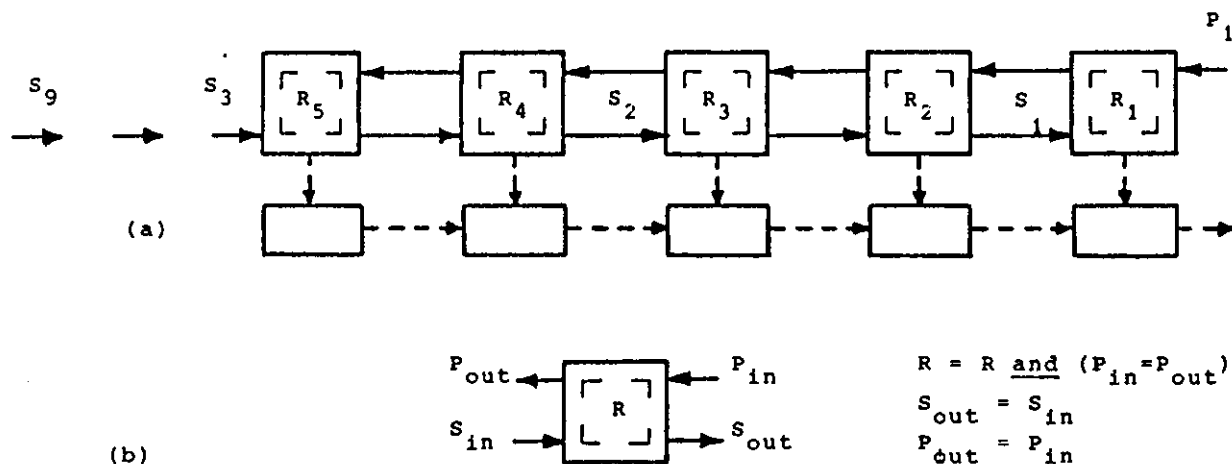


FIGURE 7.9 SYSTOLIC PATTERN MATCHER ARRAY (a), AND CELL (b), WHERE R_i 's STAY, AND S_i 's AND P_i 's MOVE SYSTOLICALLY IN OPPOSITE DIRECTIONS

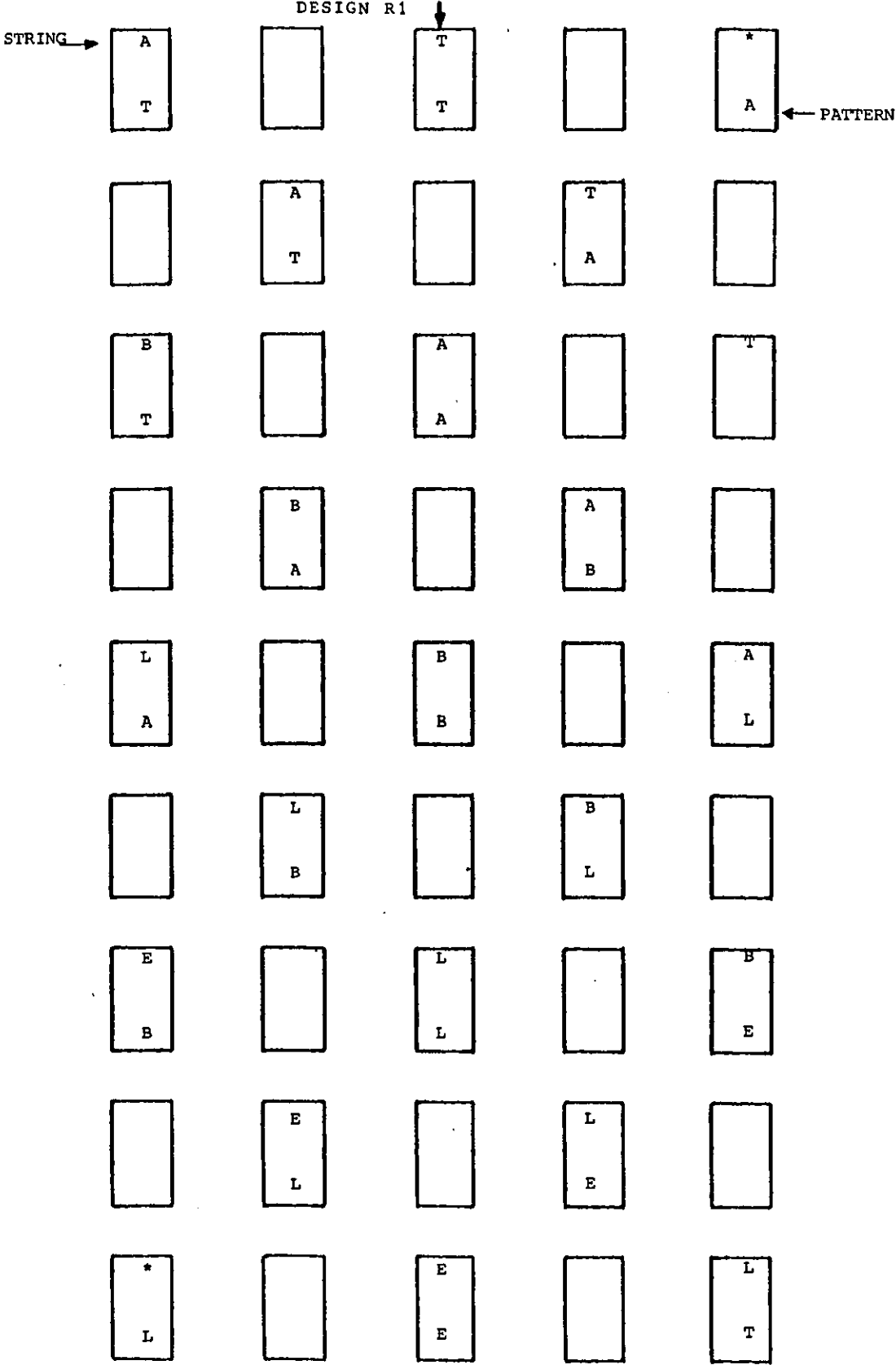
Each cell of the systolic array compares two characters and accumulates a temporary result. On each beat, every character moves from one cell to its neighbouring one (see Figure 7.9). In order to make sure that every pair of characters meet rather than just pass each other at a cell level, they are separated by one cell so that alternative cells are idle. Figure 7.10 traces the data/computation of the hard-systolic design when searching for all the occurrences of a given pattern string "TABLE" in the string defined below:

Pattern: TABLE

String: ... WITH A TABLE OR STRING ...

Following the pointer in Figure 7.10, illustrates the history of the pattern matcher chip, starting when the first character of the pattern "T" is present.

FIGURE 7.10: DATA/COMPUTATION SNAPSHOT OF THE SYSTOLIC DESIGN R1



To enable a cell to output its accumulator contents and then reset it, the first character of the pattern is associated with a tag bit (not shown in Figure 7.9). A systolic output path (indicated by broken lines in Figure 7.9) allows match results to be output in the natural ordering ($R_1, R_2 \dots$, since consecutive P_1 's are well synchronised - i.e. separated by two cycle times.

The problem with the above design is its poor performance since only one-half of the cells are doing useful work at any time. To fully exploit the potential throughput of this design, Foster and Kung suggested that two pattern matching problems could be interleaved on the same systolic array, however this implies that cells in the array must be considerably modified in order to support the interleaved processing.

Alternatively, if the pattern and the text streams move in the same direction but at different speeds, all the cells would be used efficiently. For example, if the two streams move from left to right systolically but the S_1 's move twice as fast as the P_1 's, design R_2 , illustrated in Figure 7.11, is obtained. In this case, each P_1 stays inside every cell it passes for one extra cycle, thus taking twice as long to move through the array as any S_1 . Compared to the first design, this design has the advantage that all cells work all the time, but it requires an additional register in each cell to temporarily store a pattern character.

Both designs were proved correct by running a simulation program for each one of them. The corresponding systolic programs 7.1 and 7.2 are in Appendix D.

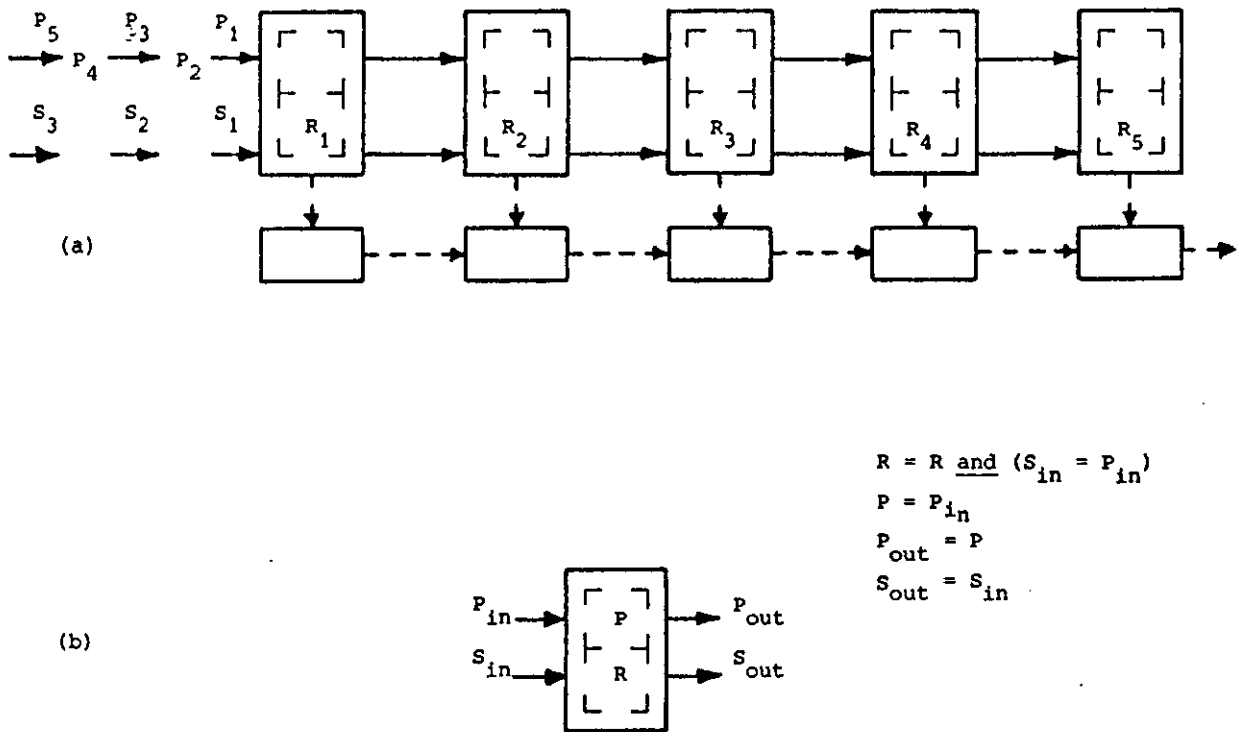


FIGURE 7.11: SYSTOLIC PATTERN MATCHER ARRAY (a) AND CELL (b), WHERE R_i 's STAY, AND S_i 's AND P_i BOTH MOVE IN THE SAME DIRECTION BUT AT DIFFERENT SPEEDS¹

7.5.2 SOFT-SYSTOLIC DESIGNS

In this section we shall present several soft-systolic designs for the pattern matching problem, which is defined as follows:

Given a pattern string of characters of length k $\{P_1P_2...P_k\}$ and a text string of length n $\{S_1S_2...S_n\}$,

Search all occurrences of the pattern in the input text string of characters. In other words, the problem consists of finding a character location i such that

$$(P_1 = S_{i+1-k}) \text{ and } (P_2 = S_{i-k}) \text{ and } \dots (P_k = S_i)$$

In order to avoid backing up the input text stream in the case of a mismatched character, k sub-strings are allowed to be compared in parallel. Consequently the pattern matching problem becomes compute-bound since each S_i fetched from the memory is used by the k cells. However, if each S_i is input from memory every time it is required (i.e. k times), then when k is large, the memory bandwidth becomes a bottleneck which might prevent any high-performance solution. As mentioned earlier, a systolic array for the pattern matcher problem resolves this bottleneck by making multiple use of each S_i . Based on this principle, several alternative designs for the pattern matching problem are described below. For simplicity we assume $k = 5$.

1) Soft-systolic pattern matcher with broadcasting

Obviously, one way to make multiple use of a single input string character, once brought from memory, is to broadcast it. In particular, if an S_i is broadcast to all cells simultaneously through separate data channels, then the same element can be consumed by the k processing cells. In the following, we shall present two different designs, B1 and B2, based on broadcasting the input characters.

The soft-systolic design B1, whose array and cell definition are illustrated in Figure 7.12, assumes that the text characters are broadcast, the pattern characters stay and the results R_i move systolically. More explanatory, the pattern characters are preloaded to the cells, one at each cell and remain at the cell throughout the entire string processing. The partial results R_i move systolically from cell to cell in the left-to-right direction.

At the start of each cycle, each S_i is broadcast to all the cells and each R_i , initialised to TRUE (i.e. 1) enters the left-most boundary cell. During cycle one, the result of a character comparison between P_1 and S_1 (i.e. $P_1 = S_1$) is accumulated in R_1 and during cycle two ($P_1 = S_2$) and ($P_2 = S_2$) are accumulated to R_2 and R_1 at the first and second cells, from the left, respectively and so on. A data flow/computation snapshot of the soft-systolic array is illustrated in Figure 7.13 where '*' represent any previous result which might be of no interest at this stage. As is shown in Figure 7.13, a result is output every cycle and an occurrence of the pattern is found after 7 cycles. A similar design of this soft-systolic pattern matcher array was previously proposed for chip implementation in [Mukhopadhyay, 1979].

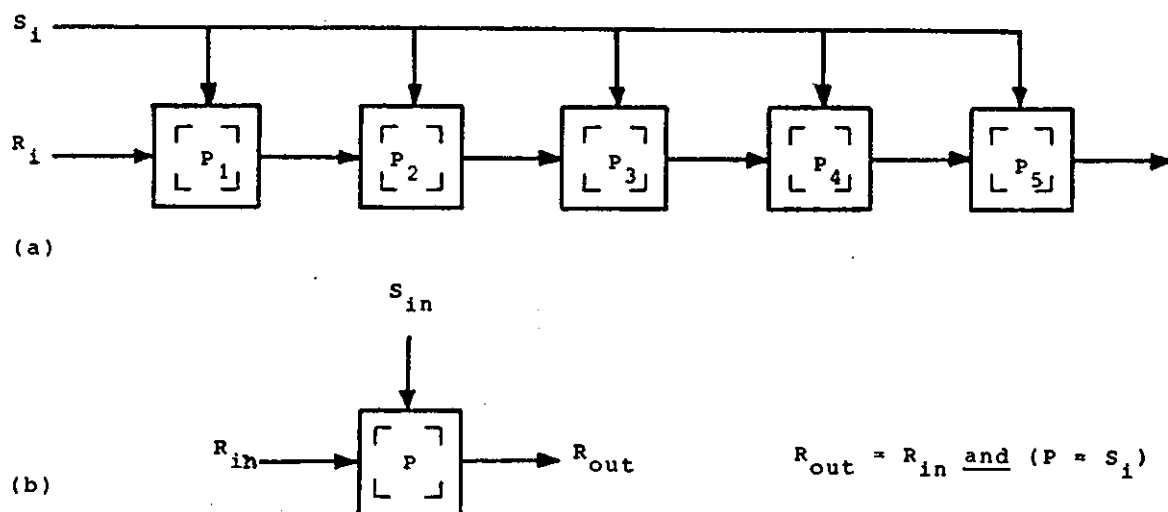


FIGURE 7.12: SOFT-SYSTOLIC PATTERN MATCHER ARRAY (a) AND CELL (b) WHERE S_i 's ARE BROADCAST, P_i 's STAY AND R_i 's MOVE SYSTOLICALLY

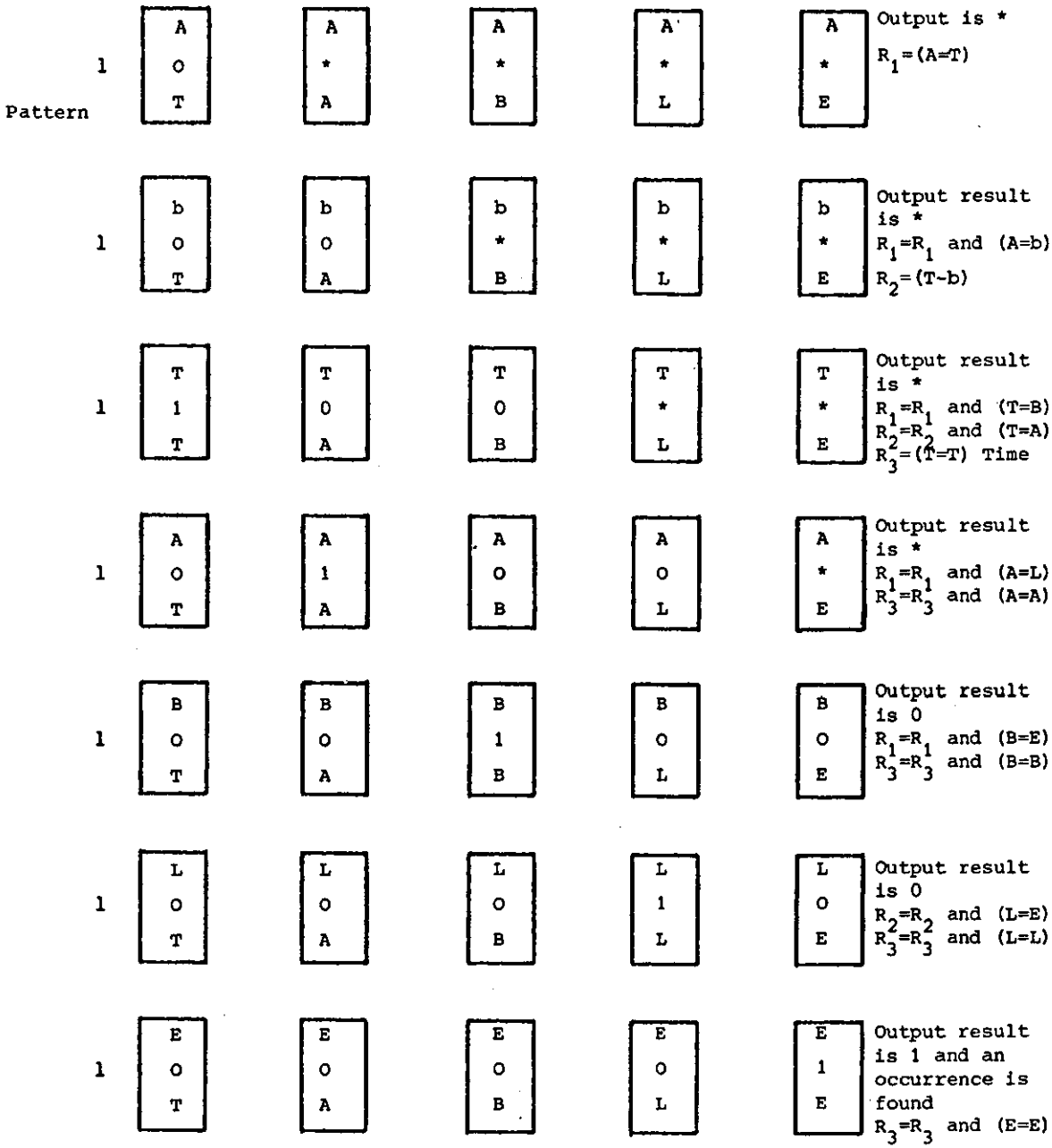


FIGURE 7.13: DATA FLOW/COMPUTATION IN THE SOFT-SYSTOLIC DESIGN B1

String: ... WITH A TABLE OR A TREE ... ,

Pattern: TABLE

Design B1 is simulated on the Balance 8000 using the OCCAM language and the soft-systolic program is reported in Appendix D, program 7.3. The program was tested on a specific example which proved satisfactory.

A second alternative design, also based on broadcasting the input text characters, is design B2 (see Figure 7.14) where the pattern characters move and the results stay. Each result R_i stays at a cell to accumulate its k terms while the pattern characters circulate around the array of cells and the first character P_1 is associated with a tag bit that signals the accumulator to output and resets its contents. Consequently, a final and correct result value R_i is output from a cell every cycle.

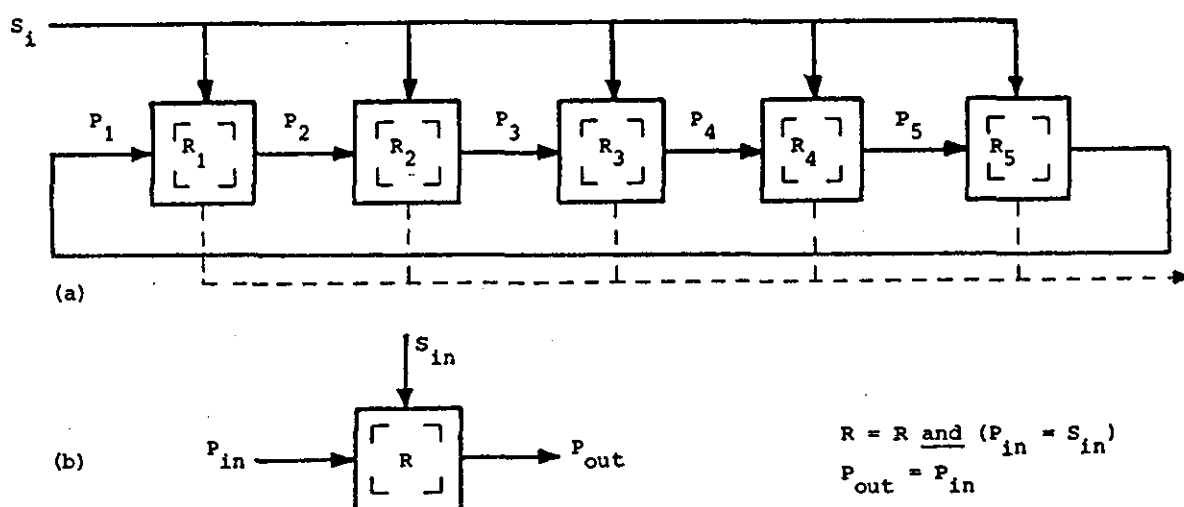


FIGURE 7.14: SOFT-SYSTOLIC PATTERN MATCHER ARRAY (a) AND CELL (b), WHERE S_i 's ARE BROADCAST, R_i 's STAY, AND P_i 's MOVE SYSTOLICALLY

Design B1 is preferred than that of B2 because it has the advantage of not requiring separate buses, one from each cell, denoted by dashed lines in Figure 7.14, for collecting outputs from individual cells. Also, the bus used in B2 to move around patterns is much wider than that used for moving results (a logical value may require only a single bit).

The correctness of this design is proved by simulating its soft-systolic array using the OCCAM language (see Appendix D, program 7.4). We also reported in Figure 7.15 a snapshot of the data flow/ computation of the array for a specific example. In this example an occurrence of the pattern 'TABLE' is found after 7 cycles.

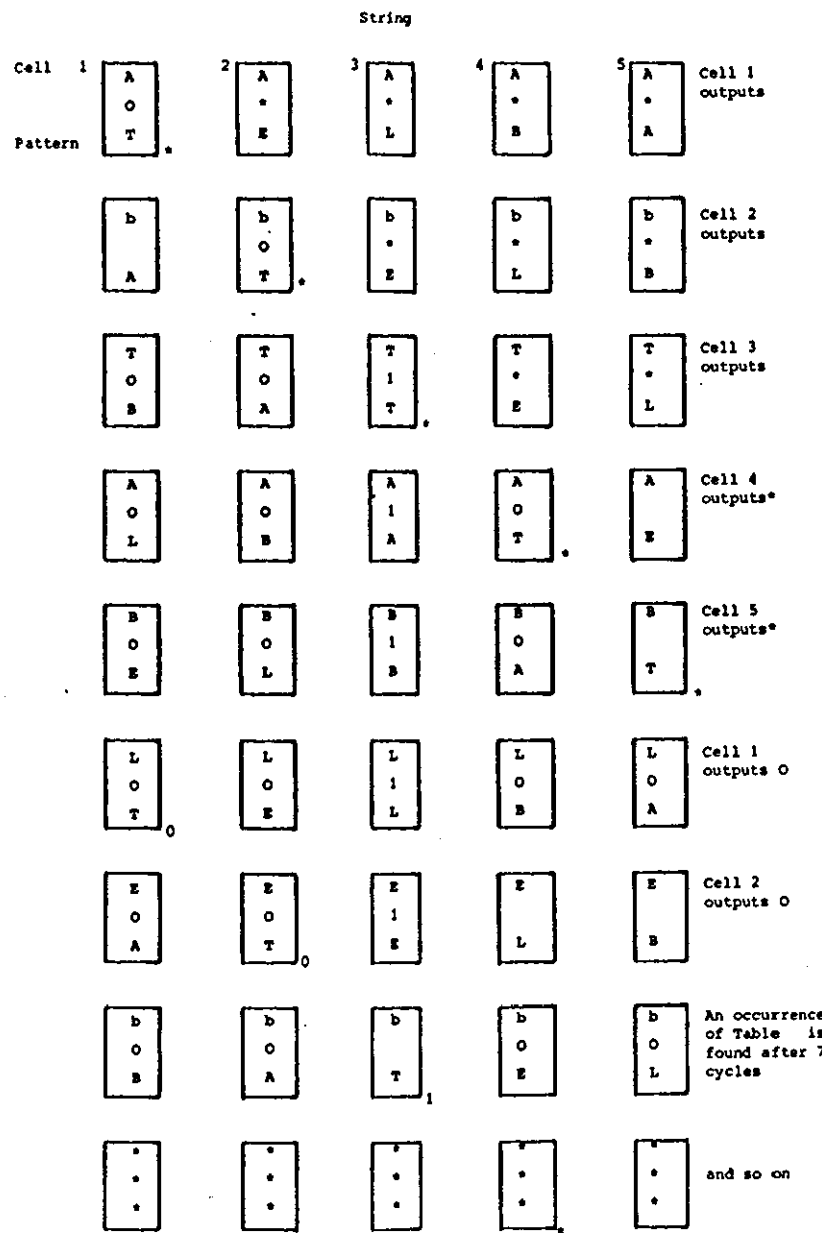


FIGURE 7.15 : DATA/COMPUTATION SNAPSHOT OF THE SYSTOLIC DESIGN B2

ii) Soft-systolic design when results are fanned-in

Each cell in the above two soft-systolic designs, performs two separate functions - it compares characters of the pattern and string and updates and outputs the match results. These two functions could be divided between two separate modules so that there are two different, but simpler, cells which are called the comparator and accumulator cells in the array.

In designing an accumulator, one could have several different alternatives. For instance, depending on the length of the array we could have a single accumulator that collects all the k individual partial comparison results and outputs a match result, or an array of comparators or a tree structure. We shall describe a soft-systolic design F1 and F2 for the first and third alternatives respectively in the following paragraphs. The second possibility was used in the hard-systolic design of Foster and Kung.

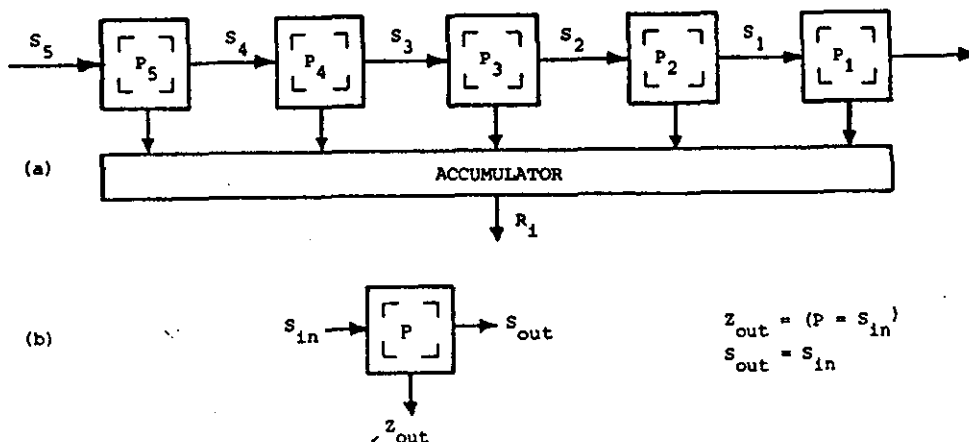
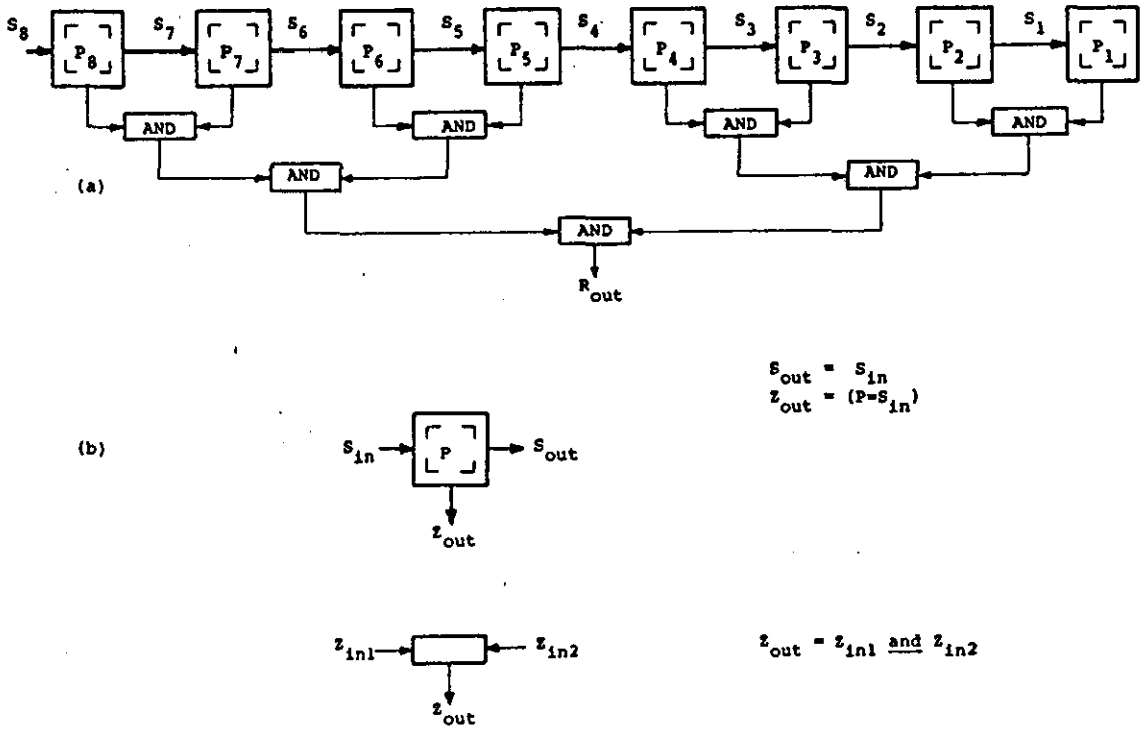


FIGURE 7.16: SOFT-SYSTOLIC PATTERN MATCHER ARRAY (a) AND CELL (b), WHERE P_i 's STAY, s_i 's MOVE SYSTOLICALLY AND R_1 's ARE FORMED THROUGH THE FAN-IN OF RESULTS FROM ALL THE CELLS

By viewing the text string as sliding over the pattern string which is supposed to be fixed in space then the pattern matching problem becomes one which compares characters of the given pattern with those of the sub-string overlapping with the pattern. Based on this view, a soft-systolic design F1, as illustrated in Figure 7.16, is suggested. The pattern characters are preloaded to the cells and stay there until the complete text processing. For a large number of cells, the accumulator can be implemented as a pipelined AND tree (see Figure 7.17).

The correctness of both designs F1 and F2 which use fan-in techniques, were successfully demonstrated through simulation on the Sequent Balance 8000. Their corresponding soft-systolic simulation

FIGURE 7.17: SOFT-SYSTOLIC PATTERN MATCHER ARRAY (a), COMPARATOR CELL (b) AND AND CELL, WHERE P_i 's STAY, S_i 's MOVE SYSTOLICALLY AND R_i 's ARE FANNED-IN THROUGH A PIPELINED AND TREE



programs are reported in Appendix D, programs 7.5 and 7.6 respectively.

7.5.3 CONCLUSIONS

Due to shortage of time, only a limited number of systolic designs have been presented and then simulated to be proved correct. This does not mean that all the possible systolic designs for the pattern matching problem have been thoroughly exhausted. For instance, it is possible to have another set of systolic designs where all the three different streams, the pattern and the text string, as well as the result stream, move systolically. Also, it could be advantageous to include enough memory storage and a limited logic control inside each cell so that the whole pattern string could be stored in each cell. With such a feature, the end result will be the design of a single pattern matcher cell which consists of an implementation of some simplified form of the Brute Force pattern matching algorithm. Once again, an array of, at least k , cells of this type will be necessary in order to avoid the burden of backing up the input text string every time a mismatch between two characters occurs. Another possible set of systolic designs for the pattern matching problem is to combine the broadcasting and fan-in techniques which, taken together, will allow the maximum use of each input and output result.

From the above set of possible additional designs, one can immediately notice that, once one systolic design is obtained a set of other systolic designs could be easily derived. The crux of the problem here is to fully understand precisely the advantages and disadvantages of each design so that an appropriate systolic algorithm is selected for a given environment. For example, it is

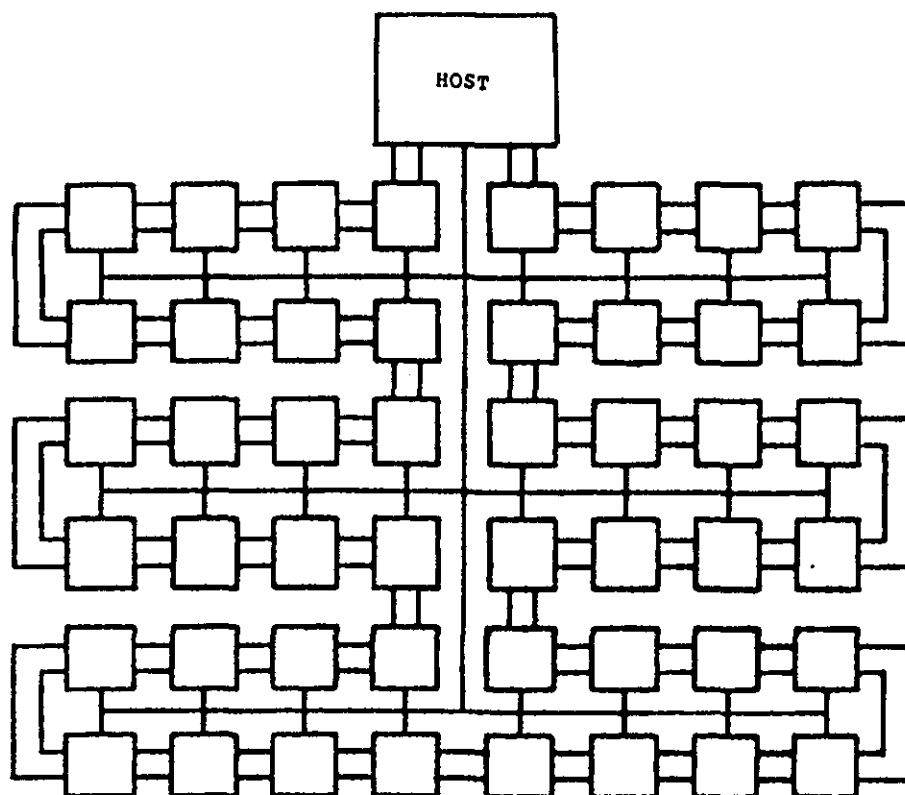


FIGURE 7.18 : USE OF THE FOLDING TECHNIQUE TO SOLVE THE DATA SKEW WHEN BROADCASTING.

useful to know that design B1 is preferred to that of B2 because of the length of the bus required for moving the pattern around the cells.

With the current VLSI technology the only selected design for chip implementation is the design R2 since it satisfies all the systolic constraints for efficient chip manufacturing. However, progress in this field is already underway and indicates that the days of ad-hoc designs are numbered. Several transformational approaches based on more flexible (but controlled) attitudes, (re-timing, replacement or synthesis operations [Megson 1987]) to defining new systolic schemes have been formally suggested. Other techniques to resolve the problem of clock skews include the use of the folding technique as indicated in Figure 7.18. This is particularly important for the designs based on broadcasting/fan-in schemes since it means that

longer patterns can be solved. Further the efforts of Leiserson and Saxe have generated new ways of converting soft-systolic designs involving broadcasting and unbounded fan-in into pure-systolic systems without broadcasting [Leiserson 1981].

Chapter 8

SUMMARY AND CONCLUSIONS

In this thesis we have studied several important non-numerical algorithms for parallel computers and VLSI systolic processor arrays. In particular, these algorithms were investigated under the framework of either being suitable for execution on asynchronous multiprocessor systems (MIMD computers) or, due to the recent rapid advance of VLSI circuitry, of being suitable for direct hardware implementation.

In the first three introductory chapters, a brief and disciplined state-of-the-art survey was compiled with up-to-date information on the parallel computing environment. This survey was complemented by the contents of Chapter 7, where we discussed the VLSI technology and its impact on the computing environment.

More analytically, in Chapter 1, we have discussed the main motivations that led to the "parallel way of thinking" and presented several different forms of exploiting this novel idea. Although several attempts (at least three of them were presented in this thesis) have been made to classify these various architectural designs, none of them seems to succeed in providing a clear distinction between classes since sometimes the intersection of two classes is not empty.

Of the architectures designed for highly parallel processing we presented the pipelined and data flow computers. Two noteworthy examples of the pipelined vector processors are the CRAY-1 from Cray Research Inc. and the CYBER 205 from Control Data Corporation. The performance of these computers is dramatically increased when more than one pipelined vector operation can be chained together, providing an added measure of concurrency. One of the fastest data flow computers ever built is the Manchester dataflow machine.

In Chapter 2 and due to the considerable interest of the industrial, governmental and university institutions, we discussed the SIMD and MIMD architectures. For the SIMD class we have surveyed its two major sub-classes: the Associative processors and the Array processors. With respect to the MIMD architecture, a particular reference was made to the TI Neptune system and the Sequent Balance 8000 computer, both sited in the Department of Computer Studies, at Loughborough University of Technology, on which the bulk of the experimental work contained herein was carried out.

Since the compilation of our survey on some of the implemented high-speed parallel computers (i.e. supercomputers) was made, a few interesting new computer architectures, worth reporting here, have been developed. We felt that we ought to bring our state-of-the-art survey on supercomputers by including a short account of some of the recent developments in this area.

As an improved version to CRAY-1, one of the most popular vector processors, the CRAY RESEARCH group introduced the CRAY X-MP computer in 1983. A year later, the CRAY-1 was taken out of production. Compared with its predecessor, the CRAY X-MP features a clock cycle time of 9.5 nsec - 'nanoseconds' (instead of 12.5 nsec), improved memory bandwidth, an increase in the maximum memory size (up to 16 Mwords) and the possibility of having one, two, or four pipelined vector processors.

Since the pipelined processors are able to cooperate on a single computation (i.e. multitasking) the X-MP is a tightly-coupled multiprocessor. Parallel application running on the CRAY X-MP have indicated a speed increase of 1.8 to 1.9 times over an uniprocessor

X-MP execution times while speed increases of 3.5 to 3.8 times have been obtained with the four-processor X-MP multiprocessor computer.

The CRAY Research Inc is currently developing a silicon-based CRAY X-MP successor that will use internally designed VLSI-chips [Thompson 1986]. Other newly built supercomputers include the following: the Hitachi S-810, the first supercomputer Array Processor ever built by Hitachi ltd, [Odaka et al, 1986], Fujitsu's supercomputer FACOM which is a vector processor system [Miura 1986] and the NEC supercomputer SX system which is capable of 1.3 gigaflops [Watanabe et al 1986].

In the MIMD multiprocessor class of computers we give the example of the CYBERPLUS supercomputer architecture which can be configured with as many as 64 processors [Allen 1986]. The processors which are connected together in a circular ring network for efficient data and control flows, are capable of cooperating together towards the execution of a single job. In addition, parallelism is also introduced within each processor by allowing the fifteen functional units which are connected via a crossbar switch, to operate concurrently. A single CYBERPLUS processor can provide up to 40 times the performance of a CYBER 170/835 in 64-bit floating point applications and even higher performance in integer applications.

Finally, the Sequent Balance 8000 system at Loughborough University has been upgraded. The number of processors has been increased from 6 to 10 processors and several additional parallel programming features, such as the data-partitioning and the function-partitioning have also been incorporated. To date, these additional software facilities are under extensive investigation to determine their potential advantages in exploiting parallelism.

In Chapter 3 we reported on the programming tools and algorithms that exploit the parallel hardware potential parallelism. In particular, concurrent programming languages motivations and general concepts for parallel processing were discussed. Various methodological design and analysis aspects of parallel algorithms that could be mapped onto different architectures were also included.

It has been noted, that in general, parallel programming is more complex than uniprocessor programming, and this has led to the parallelism being concealed on most existing MIMD computers. Therefore the search for various techniques of achieving high-speed performance with affordable reliability and cost is still a major topic of interest.

In fact, the techniques for programming these MIMD computers for efficient parallel operations are much less developed than the corresponding techniques for SIMD systems. However this does not imply that the class of problems suitable for the former type of computers can be easily implemented.

The problem which arises here is to make sure that each one of the P activated processors gets its share of task processing while maintaining some sort of cooperation between them. In order to make the multiprocessor system effective it is vital that the speed increase is substantial, hopefully of $O(P)$, in comparison with the smallest possible sequential time-complexity achievable for the same problem when solving it by one of the relatively 'best' considered existing methods.

One way of achieving this is to pay considerable attention to the problem of minimising the synchronisation operations performed by the P involved processors and the amount of data sharing amongst them. These two factors are directly dependent upon the overall computational scheduling.

The performance analysis of a parallel algorithm, although it can be more complex as the algorithm gets more complicated, has a two-fold advantage. First, it can help one to understand better the algorithm and sometimes to reveal any necessary further improvements, and second it constitutes, in the case of a good agreement with the experimental results, a validated theoretical projection for the algorithm to be run on any MIMD multiprocessor system with more than P processors.

An extensive study of the parallel searching problems were carried out in Chapter 4. Several parallel versions, based on different ways of allocating subsets to processors for the Parallel Sequential (PS) and the Parallel Binary (PB) searching algorithms were presented. For the performance analysis of these algorithms, as well as for the Parallel Jump searching algorithms (PJ), a key comparison based analytical model was extensively and successfully used. In particular, we were able to show that PS, version 1.0 was capable of achieving superlinear speed-ups, while version 2.0 only reached a linear speed-up. These were supported by several runs performed on the Balance 8000 system.

Due to the fact that the binary search method, when implemented in parallel, failed on two occasions (i.e. versions 1.0 and 2.0), to equally partition the bulk of work amongst the P processors, the PB was conclusively not accepted to be suitable for processing on an

MIMD type of computer. PB version 3.0, although succeeding in dividing the total amount of work equally between the processors, was also discarded since it introduced a large fraction of synchronisation overheads.

The third part of this chapter was the parallel implementation of the jump searching method and many of its variants - i.e. the two-level simple and the two-level fixed jump searching algorithms. Generally, the parallel implementation of these algorithms proved to be successful. However, due to the small number of operations as the jump size increased, the performance of the two level fixed jump search algorithm suffered considerably such that it was not efficient to run the algorithm with more than four processors. In conclusion, the parallel jump searching methods are more efficient with larger files.

A complete performance exploitation of both the Neptune and Balance MIMD systems were presented in Chapter 5, by implementing several parallel string pattern matching algorithms using the powerful 'divide-and-conquer' technique. The experimental results, reported in tabular form showed that, in general, many of the parallel pattern matching algorithms are well suited for MIMD implementations.

In Chapter 6, two new parallel sorting algorithms: Parallel Bounded-Partitioned and Parallel Range-Partitioned Sorting Algorithms (abbreviated respectively by PBPS and PRPS) were developed and analysed. Unlike the Parallel Quicksort (PQ) algorithm, both new algorithms have the potential of creating P independent subsets in parallel with a time-complexity for partitioning proportional to n , (n being the set size). However,

they both require larger memory storage than that of the Parallel Quicksort method to partition the original set of numbers.

The theoretical performance model of the four Parallel Sorting algorithms (including the Parallel Quicksort-Merge - 'PQM'), which was based on the number of key comparisons was validated by running these algorithms on the Sequent Balance 8000 system. In general, the experimental results were in close agreement with the theoretical results.

Chapter 7 has concentrated mainly on the introduction of some soft-systolic designs for the pattern matching problem, and their subsequent simulation using the OCCAM language. Some alternative designs were also considered which were only possible when some of the constraints as imposed by the VLSI technology were relaxed. In order to relate this chapter with the previously presented chapters, it was decided that Chapter 7 should be, at least conceptually, organised into two main parts.

In the first part which constitutes a complement to the survey on parallel computer architectures, introduced in Chapters 1 and 2, we have presented the VLSI technology as a substantial contender to the achievement of very high-performance, cost-effective computing systems for the future decades. We have also presented its fundamental concepts such as regularity, planarity, use of pipelining and concurrency, in designing special-purpose and general-purpose computing structures.

For the special-purpose class of VLSI-oriented systems we have established two main contenders which are the systolic arrays as suggested by H.T. Kung and the wavefront arrays resulting from the

work of Y.S. Kung. Although these systems are cost-effective, they are however specially designed for one particular problem. In order to increase flexibility, the general-purpose computing structures such as the WARP, built by H.T. Kung and the CHIP of L. Snyder can be used to solve a predefined set of algorithms.

Following these substantial benefits, a research program was initiated in the Department of Computer Studies, at Loughborough University to investigate the Instruction Systolic Array - 'ISA'. This is a novel idea which consists of broadcasting along with the data, the instruction that is performed on it. A primitive assembler/compiler for a special language, the Replicated Instruction Systolic Array Language - 'RISAL' was also devised. Using such a language it was possible to design simple test examples which could be investigated thoroughly to first determine major extensions to the language itself and possibly to highlight potential problems within the ISA machine.

As far as the pattern matching problem is concerned, the Karp-Rabin algorithm which is a compute-bound problem is well suited for VLSI implementation. Its hardware algorithm would require as much as k multiply-and-add (IPS) cells to compute the hash function of both the pattern and the current substring, and a single comparator cell at the boundary of the array to compare these two hash values. The systolic design should be straightforward since it is similar to that of the pattern matcher chip.

In conclusion, we should stress our firm vision that the systolic computing paradigm will play a major role in future supercomputing, especially for those compute-bound problems. Furthermore, most existing computing networks will be systematically converted into

systolic or wavefront arrays following the already established procedures. This fact will certainly boost the development of sophisticated hardware and advanced software for the supercomputers of the future.

REFERENCES

REFERENCES

ABEL, N.E., BUDNIK, P.P., KUCK, D.J., MURAOKA, Y., NORTHCOTE, R.S. and WILHELMSON, R.B. [1969]:

"TRANQUIL: A Language for an Array Processing Computer", AFIPS Conf. Proc. 34, 1969, pp 57-75.

AHO, A.V. and CORASICK, M.J. [1975]:

"Efficient String Matching: an Aid to Bibliographic Search", Communications of the ACM, June 1975, Vol. 18, No. 6, pp 333-340.

ALLEN, G.R. [1986]:

"Parallel Processing System - CYBERPLUS", Supercomputers, Class VI Systems, Hardware and Software, S. Fernback (ed), North-Holland, pp 169-181.

ANDERSON, G.A. and JENSEN, E.D. [1975]:

"Computer Interconnection Structures: Taxonomy Characteristics and Examples", Computing Surveys, Vol. 7, No. 4, Dec. 1975, pp 197-213.

ANDERSON, G.A. and KAIN, R.Y. [1976]:

"A Content-Addressed Memory Design for Data Base Applications" in Proc. 1976 International Conf. on Parallel Processing, IEEE, New York, 1976, pp 191-195.

ANDERSON, J.P. [1965]:

"Program Structures for Parallel Processing", Communications of the ACM, Vol. 8, No. 12, 1965, pp 786-788.

ARVIND, and GOSTELOW, K.P. [1982]:

"The U-Interpreter", IEEE Computers, Feb. 1982, pp 42-49.

ARVIND, and THOMAS, R.E. [1980]:

"I: Structures: An Efficient Data Type for Functional Languages". MIT/LCS/TMN-178, Sept. 1980.

ARVIND, et al [1983]:

"A Critique of Multiprocessing Von Neumann Style". 10th ACM Architecture Symposium, 1983, pp 426-436.

BAASE, S. [1983]:

"Computer Algorithms: Introduction to Design and Analysis", Addison-Wesley Publishing Company.

BACKUS, J. [1978]:

"Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", Communication of the ACM, Vol. 21, No.8, Aug. 1978, pp 613-641.

BAER, J.L. [1976]:

"Multiprocessing Systems", IEEE Trans. Comput., Vol. C-25, No. 12, Dec. 1976, pp 1271-1277.

BARHAMI, B. [1972]:

"A Highly Parallel Computing System for Information Retrieval", in Proc. AFIPS Fall Joint Computer Conf., AFIPS Press, Montvale, NJ, 1972, pp 681-690.

BARLOW, R.H., EVANS, D.J., NEWMAN, I.A. and WOODWARD, M.C. [1981]:

"A Guide to Using the Neptune Parallel Processing System", Internal Report, Computer Studies Dept., Loughborough University, UK, 1981.

BARNES, G.H. et al [1968]:

"The Illiac IV Computer", IEEE Trans. Comput., Vol. C-17, No. 8, Aug. 1968, pp 746-757.

BASKET, F. and SMITH, A.J. [1976]:

"Interference in Multiprocessor Computer Systems with Interleaved Memory", Communications of the ACM, Vol. 19, No. 6, June 1976.

BATCHER, K.E. [1974]:

"STARAN Parallel Processor System Hardware", in Proc. AFIPS 1974 National Computer Conf., Vol. 43, AFIPS Press, Montvale, NJ, 1974, pp 405-410.

BATCHER, K.E. [1979]:

"The Massively Parallel Processor (MPP) System", Proc. AIAA Aerospace Conf, 1979, pp 93-97.

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.

BEHNKE, E.A. and ROSENBERGER, G.B. [1963]:

"Cryogenic Associative Processor", IBM Final Report, Sept. 1963.

BEKAKOS, M.P. and EVANS, D.J. [1986]:

"The Exposure and Exploitation of Parallelism on Fifth Generation Computer Systems", Parallel Computing 85, Elsevier Science Publishers BV (North Holland), pp 425-442.

BERNSTEIN, A.J. [1966]:

"Analysis of Programs for Parallel Processing", IEEE Trans. on EC, Vol. 15, No. 5, Oct. 1966, pp 757-763.

BOYER, R.S. and MOORE, J.S. [1977]:

"A Fast String Searching Algorithm". Communications of the ACM, October 1977, Vol. 20 No. 10, p 762.

BRENT, KUNG, H.T. and LUK [1983]:

"Some Linear Time Algorithms for Systolic Arrays". CMU-ROL-83 and invited paper 9th World Computer Congress, Paris, 1983.

CAMPBELL, R.H. and HABERMANN, A.N. [1974]:

"The Specification of Process Synchronisation by Path Expression". Lecture Notes in Computer Science, 16, Springer, 1974.

CHANG, D.Y. et al [1977]:

"On the Effective Bandwidth of Parallel Memories", Vol. C-26, No. 5, May 1977, pp. 480-490.

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.

CHU, Y.H. [1965]:

"A Destructive-Readout Associative Memory", IEEE Trans. Computers EC-14, Aug. 1965, pp 600-605.

COMTE, D., HIFDI, N. and SYRE, J.C. [1980]:

"The Data Driven LAU Multiprocessor System: Results and Perspective". Proc. IFIP, 1980, pp 175-180.

CONTROL Data Corporation CDC [1980]:

"Advanced Flexible Processor", 1980.

CORNELL, J.A. [1972]:

"Parallel Processing of Ballistic Defence Radar Data with PEPE", IEEE COMPCON, 1972, pp 69-72.

CORNISH, M. [1979]:

"The TI Dataflow Architectures: the Power of Concurrency for Avionics". Proc. 3rd Conf. on Digital Avionics Systems, IEEE, New York 1979, pp 19-25.

COURANZ, G.R., GERHARDT, M.S. and YOUNG, C.J. [1974]:

"Programmable Radar Signal Processing Using the RAP", in Proc. Sagamore Computer Conference on Parallel Processing, Springer-Verlag, NY, 1974, pp 37-52.

CRANE, B.A. and GITHENS, J.A. [1965]:

"Bulk Processing in Distributed Logic Memory", IEEE Trans. on Electronic Computers, Vol. EC-14, April 1965, pp 186-196.

CRANE, B.A., GILMARTIN, M.J., HUTTEN-HOFF, J.H., RUX, P.T. and SHIVELY, R.R. [1972]:

"PEPE Computer Architecture", IEEE COMPOON, 1972, pp 57-60.

CROJUT, W.A. and SOTTILE, M.R. [1966]:

"Design Techniques of a Delay-Line Addressed Memory", IEEE Trans. Computers, Aug. 1966, pp 523-534.

DAVIS, E.W. [1974]:

"STARAN Parallel Processor System Software", in Proc. AFIPS 1974, National Computer Conf., Vol. 43, AFIPS Press, Montvale, NJ, 1974, pp 17-22.

DENNIS, J.B. [1974]:

"First Version of a Data Flow Procedure Language". Computer Science, Vol. 19, Springer-Verlag, 1974, pp 362-376.

DENNIS, J.B. [1980]:

"Data Flow Supercomputer", IEEE Computer, Nov. 1980, pp 48-56.

DENNIS, J.B. and VAN HORN, E.C. [1966]:

"Multiprogrammed Computations". Communications of the ACM, Vol. 9, 1966, pp 143-155.

DENNIS, J.B., LIM, W.Y.P. and AKERMAN, W.B. [1983]:

"The MIT Data Flow Engineering Model", IFIP Proc. 1983, pp 553-563.

DIJKSTRA, E.M. [1965]:

"Solution of a Problem in Concurrent Programming Control". Communications of the ACM, Vol. 8, No. 9, Sept. 1965, p.569.

DIJKSTRA, E.M. [1968]:

"Cooperating Sequential Processes" in Programming Languages, et. by F. Genuys, IBM Paris, France, Academic Press, 1968, pp 43-112.

DINGELDINE, J.R., MARTIN, H.R. and PATTERSON, W.M. [1973]:

"Operation System and Support Software for PEPE", in Proc. 1973 Sagamore Computer Conf. on Parallel Processing, Springer-Verlag, NY, pp 170-178.

ENSLOW, P.H. [1977]:

"Multiprocessor Organisation - a Survey", Comput. Surveys, Vol. 9, No. 1, March 1977, pp 103-129.

EVANS, D.J. and WILLIAMS, S.A.

"Analysis and Detection of Parallel Processable Code". The Computer Journal, Vol. 23, No. 1, 1978, pp 66-72.

EVANS, D.J. and DUNBAR, R.C. [1982]:

"Parallel Quicksort Algorithm, Part 1: Run-Time Analysis", Int. J. Comp. Math., 12, pp 19-55.

EVANS, D.J. and DUNBAR, R.C. [1982]:

"Parallel Quicksort Algorithm, Part 2: Simulation", Gordon and Breach Science Publishers Inc., Int. Jour. Comp. Math., 12, pp 125-133.

EVENSEN, A.J. and TROY, J.L. [1973]:

"Introduction to the Architecture of 288-Element PEPE", in Proc. 1973 Sagamore Computer Conf. on Parallel Processing, Springer-Verlag, NY, 1973, pp 162-169.

EWING, R.G. and DAVIES, P.M. [1964]:

"An Associative Processor", in Proc. AFIPS 1963 Fall, Joint Computer Conf., Spartan Books Inc, Baltimore, Md, 1964, pp 147-158.

FINNILA, C.A. [1977]:

"The Associative Linear Array Processor", IEEE Trans. on Computers, Vol. C-26, No. 2, Feb 1977, pp 112-125.

FISHER, D.A. [1967]:

"Program Analysis for Multiprocessing", Burroughs Corp., May 1967.

FLYNN, M.J. [1966]

"Very High-Speed Computing Systems", Proc. of the IEEE, Vol. 54, No. 12, Dec. 1966, pp 1901-1909.

FOSTER, M.J. and KUNG, H.T. [1980]:

"The Design of Special-Purpose VLSI Chips". Computer, Vol. 13, No. 1, January 1980, pp 26-40.

FULLER, R.H. [1967]:

"Associative Parallel Processing", Proc. AFIPS Spring, Joint Computer Conf, 1967, pp 471-475.

GAINS, R.S. and LU, C.Y. [1965]:

"An Improved Cell Memory", IEEE Trans. Computers, Feb. 1965, pp 72-75.

GALIL, Z. [1979]:

"On Improving the Worst-Case Running Time of the Boyer-Moore String Searching Algorithm". Communications of the ACM, September 1979, Vol. 22, No. 9, pp 505-508.

GANDIO, J.L. and ERCEGOVAC, M.D. [1982]:;

"A Scheme for Handling Arrays in Data Flow Systems". Proc. 3rd Intl. Conf. Distributed Computing Systems, 1983, pp 235-242.

GEHRIG, E. et al [1982]:

"The CM^{*} Testbed", IEE Computer, Oct. 1982, pp 40-53.

GHANEMI, S. and EVANS, D.J. [1986a]:

"A Study of Parallel String Searching Algorithms". Int. Report, CS, No. 310, August 1986. To be published in Int. Jour. of Comp. Maths. Vol. 23, 1988.

GHANEMI, S. and EVANS, D.J. [1986b]:

"Parallel Sorting Algorithms", CS Int. Rep. No. 330, November 1986, currently being refereed by the Parallel Computing Journal.

GONZALEZ, M.J. and RAMAMOORTHY, C.V. [1969]:

"Recognition and Representation of Parallel Processable Streams in Computer Programs". Symposium on Parallel Processor Systems, Technologies and Applications, ed. L.C. Hobbs, Spartan Books, June 1969.

GOSDEN, J.A. [1966]:

"Explicit Parallel Processing Description and Control in Programs for Multi and Uni-Processor Computers", Proc. FJCC, Vol. 29, 1966, pp 651-660.

GOTTLIEB et al [1983]:

"The NYU Ultracomputer Designing an MIMD Shared Memory Parallel Machine", IEEE Trans. Computer, Vol. C-32, No. 2, Feb. 1983, pp 175-189.

GURD, J.R., KIRKHAM, C.C. and WATSON, I. [1985]:

"The Manchester Prototype Dataflow Computer", Comm. ACM, No. 1, Jan. 1985, pp 34-52.

HANDLER, W. [1982]:

"Innovation Computer Architectures - how to Increase Parallelism but not Complexity", in Parallel Processing Systems, Evans, D J (Ed), Cambridge University Press, 1982, GB, pp 1-41.

HANSEN, P.B. [1973]:

"Operating System Principles". Prentice-Hall, Englewood Cliffs, N.J., 1973.

HANSEN, P.B. [1975]:

"The Programming Language Concurrent Pascal" IEEE-TSE 1,2 June 1975, pp 199-207.

HANSEN, P.B. [1977]:

"The Architecture of Concurrent Programs", Prentice-Hall Series by Automatic Computation, N.J., 1977.

HARDING, P.A. and ROLUND, M.W. [1968]:

"A2- D Core Search Memory", in Proc. AFIPS Fall, Joint Computer Conf., Thompson Books Co, Washington DC, 1968, pp 1213-1218.

HAYAFIL, L. and KUNG, H.T. [1974]:

"Parallel Algorithms for Solving Triangular Linear Systems with Small Parallelism Parameter", Dept. of Computer Science, Carnegie Mellon University.

HAYAFIL, L. and KUNG, H.T. [1975]:

"Bounds on the Speed-up of Parallel Evaluation of Recurrences", Proc. Second USA-Japan Computer Conf., pp 178-182.

HAYES, J.P. [1978]:

"Computer Architecture and Organisation", McGraw-Hill, Kogakusha Ltd, Japan, 1978.

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. EC-15, Feb. 1968, pp 82-91.

HIGBIE, L.C. [1972]:

"The OMEN Computers: Associative Array Processors", IEEE Comp. Conf., 1972, Digest, pp 287-290.

HOARE, C.A.R. [1962]:

"Quicksort", Computer Journal, p.10.

HOARE, C.A.R. [1962]:

"Algorithm 64, Quicksort", Communications of the ACM, Vol. 4, No. 7, p.321.

HOARE, C.A.R. [1972]:

"Towards a Theory of Parallel Programming" in Operating Systems Techniques, C.A.R. Hoare and R. Perrott (eds)., Academic Press, New York, 1972.

HOARE, C.A.R. [1978]:

"Communicating Sequential Processes", Communications of the ACM, Vol. 21, No. 8, Aug., 1978, pp 666-677.

HOBBS, L.C. and THESIS, D.J. [1970]:

"Survey of Parallel Processor Approaches and Techniques", in Parallel Systems: Technology and Applications, Hobbs et al (Eds), Spartan Books, New York, 1970, pp 3-20.

HOCKNEY, R.W. and JESSHOPE, C.R. [1981]:

"Parallel Computers: Architecture: Programming and Algorithms", Adam Hilger Ltd, Bristol, England, 1981.

HWANG, K. and BRIGGS, F.A. [1984]:

"Computer Architecture and Parallel Processing", McGraw-Hill Computer Science Series.

JESSHOPE, C.R. and CRAIGIE, J. [1980]:

"Another Matrix Algorithm for the DAP", DAP Newsletter, Vol. 4, pp 7-14.

KAPLAN, A. [1963]:

"A Search Memory Subsystem for a General Purpose Computer", in Proc. AFIPS 1963 Fall Joint Computer Conf., Vol. 24, Spartan Books Inc, Baltimore, Md, 1963, pp 193-200.

KNUTH, D.E. [1969]:

"The Art of Computing, Vol. 1: Fundamental Algorithms", Addison-Wesley, Reading, Massachusetts.

KNUTH, D.E. [1973]:

"The Art of Computer Programming: Vol. 3 Sorting and Searching", Addison-Wesley Publishing Company, Reading, Massachusetts.

KNUTH, D.E., MORRIS, J.H. and PRATT, V.R. [1977]:

"Fast Pattern Matching Algorithm". SIAM J. of Computing, June 1977, Vol. 6, No. 2, pp 323-350.

KUCK, D.J. [1977]:

"A Survey of Parallel Machine Organisation and Programming", Computing Surveys, Vol. 9, No. 1, March 1977, pp 29-59.

KUCK, D.J. and MARUYAMA, K. [1975]:

"Time Bounds on the Parallel Evaluation of Arithmetic Expressions", SIAM J. Computing, 4, pp 147-162.

KUCK, D.J., LAWRIE, D.H. and SAMEH, A.M. (eds) [1977a]:

"High Speed Computer and Algorithm Organisation", Academic Press, New York, 1977.

KUCK, D.J. and STOKES, A.R. [1982]:

"The Burroughs Scientific Processor (BSP)", IEEE Trans. Comput., Vol. C-31, No. 5, May 1982, pp 363-376.

KUNG, H.T. [1976]:

"Synchronised and Synchronous 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. Conf. Very Large Scale Integration: Architecture, Design, Fabrication, California Institute of Technology, January 1979, pp 65-90.

KUNG, H.T. [1980]:

"The Structure of Parallel Algorithms", Advances in Computers, Vol. 19, pp 65-112, Academic Press, New York.

KUNG, H.T. [1984]:

"Systolic Algorithms for the CMU WARP Processor", CMUC-CSA-84-158 (7th Int. Conf. on Pattern Recognition Also).

KUNG, H.,T. and LEISERSON, C.E. [1978]:

"Systolic Arrays (for VLSI)" in Proc. Sparse Matrix Symp. (SIAM), 1978, pp 256-282.

KUNG, S.Y. [1985]:

"VLSI Array Processors", IEEE ASSP Magazine, July 1985, pp 5-22.

LAWRIE, D.H., LAYMAN, T., BAER, D. and RANDAL, J.M. [1975]:

"GLIPNIR - a Programming Language for ILLIAC IV", Comm. ACM, Vol. 18, March 1975, pp 157-164.

LEE, C.Y. and PAULL, M.C. [1963]:

"A Content Addressable Distributed Logic Memory with Applications to Information Retrieval", Proc. IEEE, 51, June 1963, pp 924-932.

LEE, C.Y. [1962]:

"Intercommunicating Cells, Basis for a Distributed Logic Computer", in Proc. AFIPS 1962, Fall Joint Computer Conf., Spartan Books Inc., Baltimore, Md, 1962, pp 130-136.

LEISERSON, C.E. and SAXE, J.B. [1981]:

"Optimising Synchronous Systems", Proc. 22nd Annual Symp. Foundations of Computer Science, IEEE Computer Society, Oct. 1981, pp 23-36.

LILLEVIK, S.L. and EASTERDAY, J.L. [1984]:

"Throughput of Multiprocessor with Replicated Shared Memories", National Computer Conference, 1984, pp 51-56.

LOESER, R. [1974]:

"Some Performance Tests of Quicksort and Descendants", Communications of the ACM, Vol. 17, No. 3, p.143.

LOVE, H.H. Jr [1973]:

"An Efficient Associative Processor Using Bulk Storage" in Proc. 1973 Sagamore Computer Conf. on Parallel Processing, Springer-Verlag NY, 1973, pp 103-112.

MARUYAMA, K. [1973]:

"The Parallel Evaluation of Matrix Expressions", IBM T.J. Watson Research Centre, York Town Heights, N.Y.

MEAD, C.A. [1981]:

"VLSI and Technological Innovations", in VLSI 81, Proceedings of the 1st International Conference on Very Large Scale Integration, Univ. of Edinburgh, August 1981, J.P. Gray (ed), Academic Press, London, UK, pp 3-11.

MEAD, C.A. and CONWAY, L.A. [1980]:

"Introduction to VLSI Systems", Addison-Wesley, Reading, Mass. 1980.

MEGSON, G.M. [1987]:

"Novel Algorithms for the Soft-Systolic Paradigm", PhD Thesis, Computer Studies Dept., Loughborough University of Technology.

MEGSON, G.M. and EVANS, D.J. [1986]:

"Simulation of Soft-Systolic Arrays with Boundary and Special Processing Elements". Computer Studies Departmental Report, August 1986, No. 309.

MIRANKER, W.L. [1971]:

"A Survey of Parallelism in Numerical Analysis", SIAM Review, Vol. 13, No. 4, 1971, pp 524-547.

MODEL 55 [1971]:

"Dual Memory Controller", Information Specification, Interdata Inc., 1971, New Jersey 07757, USA.

MUKHOPADHYAY, A. [1979]:

"Hardware Algorithms for Non-numeric Computations", IEEE Trans. Computers, Vol. C-28, No. 26, June 1979, pp 384-394.

MULLER, D.E. and PREPARATA, F.P. [1976]:

"Restructuring of Arithmetic Expressions for Parallel Evaluation", Journal of the ACM, Vol. 23, No. 3, July 1976, pp 534-543.

MURTHA, J. and BEADLES, R. [1964]:

"Survey of the Highly Parallel Information Processing Systems", Prepared by the Westinghouse Electric Corp., Aerospace Division, ONR Report No. 4755, Nov. 1964.

NEWMAN, I.A., STALLARD, R.P. and WOODWARD, M.C. [1984]:

"Combined Resource-Sharing Algorithm", IEEE Proceedings, Vol. 131, Pt. E, No. 2, March 1984, pp 55-60.

ODAKA, T., NAGASHIMA, S. and KAWABE, S. [1986]:

"Hitachi Supercomputer S-810 ARRAY PROCESSOR SYSTEM", Supercomputers, Class VI Systems, Hardware and Software, S. Fernabck (ed), North-Holland, pp 137-151.

PATEL, J.H. [1981]:

"Performance of Processor-Memory Interconnections for Multiprocessors", IEEE Trans. Comput., Vol. C-31, No. 10, Oct. 1981, pp 771-780.

PATEL, J.H. [1982]:

"Analysis of Multiprocessors with Private Cache Memories". IEEE Trans. on Computers, Vol. C-31, No. 4, Apr. 1982, pp 296-304.

PERROTT, R.H. [1978]:

"Scientific Computing in the 1980s: Programming Language", Nat. Bureau Standards 17th Ann. Tech. Symp. on Tools for Improved Computing, 1978, pp 65-69.

RAMAMOORTHY, C.V. and GONZALEZ, M.J. [1969]:

"A Survey of Techniques for Recognising Parallel Processable Streams in Computer Programs", Fall Jt. Computer Conference 1969, Work Supported by NASA Grant NGR 44-012-144.

RAMAMOORTHY, C.V. and LI, H.F. [1977]:

"Pipeline Architecture", Computer Survey, Vol. 9, No. 1, March 1977, pp 61-102.

REDDAWAY, S.F. [1973]:

"DAP - a Distributed Array Processor", Proc. of First Symp. on Computer Architecture, 1973, pp 61-65.

RICHARD, J., SWAN, R.J., BECHTOLSHEIN, A., LAI, K.W. and OUSTERHOUT, J.K. [1977]:

"The Implementation of the Cm* Multi-Multiprocessor", National Computer Conference, 1977, pp 637-646.

RUDOLPH, J.A. [1972]:

"A Production Implementation of an Associative Array Processor: STARAN", in Proc. AFIPS 1972, Fall Joint Computer Conf., Vol. 41, Pt. 1, AFIPS Press, Montvale, NJ, 1972, pp 229-241.

SATYANARAYANAN, M. [1980]:

"Commercial Multiprocessing Systems", IEEE Computer, Vol. 13, No. 5, May 1980, pp 75-98.

SEDGEWICK, R. [1984]:

"Algorithms", Addison-Wesley Publishing Company, Reading, Massachusetts.

SEIGEL, H.J.

"Interconnection Networks for SIMD Machines", IEEE Computer, June 1979, pp. 57-65.

SHAPIRO, E. [1984]:

"Systolic Programming: a Paradigm of Parallel Processing". Department of Applied Mathematics, the Weizmann Institute of Science, Rehovot 76100, Israel, Draft, Sept. 1984.

SHOUMAN, W. [1960]:

"Parallel Computing with Vertical Data", in Proc. 1960 Eastern Joint Computer Conf., NY, 1960, pp 111-115.

SHORE, J.E. [1973]:

"Second Thoughts on Parallel Processing", Comput. Elect. Eng., 1973, pp 95-109.

SLADE, A. and McMAHON, H.O. [1957]:

"A Cryotron Catalog Memory System", Proc. 1956 Eastern Joint Computer Conf., American Inst. of Electrical Engineers, New York, 1957, pp 115-120.

SLOTNICK, D.L., BORCH, W.C. and McREYNOLDS, R.C. [1962]:

"The SOLOMON Computer", 1962 Fall Joint Computer Conf., American AFIPS Proc. Vol. 22, Washington, Spartan, DC 1962, pp 97-107.

SLOTNICK, D.L., BORCH, W.C. and McREYNOLDS, R.C. [1963]:

"The SOLOMON Computer - a Preliminary Report", Proc. 1962 Workshop on Computer Organisation, Washington, Spartan DC, 1963, pp 66-92.

SLOTNICK, D.L. [1967]:

"Unconventional Systems", Proc. AFIPS Spring, Joint Computer Conf., 1967, pp 477-481.

SMITH, B.J. [1981]:

Architecture and Applications of the HEP Multiprocessor Computer Systems", Proc. SPIE Conf. Real Time Signal Processing 4, Vol. 298, 1981.

SQUIRE, J.S. [1963]:

"A Translation Algorithm for a Multiprocessor Computer", Proc. 18th ACM Natl. Conf. 1963.

STEVENS, K.G. Jr [1975]:

"CFD - a FORTRAN-like language for the ILLIAC IV, SIGPLAN Vol. 10, No. 3, 1975, pp 72-80.

STONE, H.S. [1967]:

"One-Pass Compilation of Arithmetic Expressions for a Parallel Processor", Comm. ACM, Vol. 10, No. 4, April 1967, pp 220-223.

STONE, H.S. [1968]:

"Associative Processing for General Purpose Computers through the use of Modified Memories" in Proc. AFIPS 1968 Fall, Joint Computer Conference, Thompson Books Co, Washington DC, pp 949-955.

STONE, H.S. [1971]:

"Parallel Processing with the Perfect Shuffle", IEEE Trans. on Computers, Vol. C-10, Feb. 1971, No. 2, pp 153-161.

STONE, H.S. [1973]:

"Problems of Parallel Computation". In Complexity of Sequential and Parallel Algorithms, edited by J.F. Traub, Academic Press, New York, 1973, pp 1-16.

STONE, H.S. [1980]:

"Parallel Computers" in Introduction to Computer Architectures, Stone, H S (Ed), SRA, Chicago, USA, 1975, pp 318-374.

SWAN, R.J., FULLER, S.H. and SIEWIOREK, D.P.

"Cm* - a Modular Multi-Microprocessor", National Computer Conference, 1977 pp 637-646.

TANG, C.Y. and LEE, R.C. [1984]:

"Optimal Speed-up of Parallel Algorithms Based upon Divide-and-Conquer Strategy", Information Science, Vol. 32, pp 173-186.

TEXAS Instruments, II, IV and VI:

"Texas Instruments DX10 Operating System", Reference Manual, Vols. II, IV and VI, Computer Studies Dept., Loughborough University, UK.

THOMAS, C.D. and KUNG, H.T. [1977]:

"Sorting on a Mesh-Connected Parallel Computer", CACM, Vol. 20, No. 2, 1977, pp 263-271.

THOMPSON, J.R. [1986]:

"The CRAY-1, the CRAY X-MP, the CRAY-2 and Beyond: the Supercomputers of CRAY Research", Supercomputers, Class VI Systems, Hardware and Software, S. Fernback (ed), North-Holland, pp 69-81.

THURBER, K.J. and WALD, L.D. [1975]:

"Associative and Parallel Processors", Computing Surveys, Vol. 7, No. 4, Dec. 1975, pp 215-255.

UNGER, S.H. [1958]:

"A Computer Oriented Toward Spatial Problems", Proc. IRE, Oct. 1958, pp 17-44.

VICK, C.R. and MERWIN, R.E. [1973]:

"An Architecture Description of a Parallel Processing Element", in Proc. 1973 International Workshop on Computer Architecture.

WANG, P.S. and LIU, M.T. [1980]:

"Parallel Processing of High-level Language Programs", in Proceedings of the 1980 Intern. Conf. on Parallel Processing, Computer Society Press, 1980, pp 17-25.

WATANABE, T., KATAYAMA, H. and IWAYA, A. [1986]:

"Introduction of NEC Supercomputer SX System", Supercomputers, Class VI Systems, Hardware and Software, S. Fernback (ed), North-Holland, pp

WIDDOES, L.C. Jr. [1980]:

"The S-1 Project: Developing High-Performance Digital Computers", IEEE COMPOON Proc. Spring 1980, pp 282-291.

WILSON, D.E. [1972]:

"The PEPE Support Software System", IEEE COMPOON, 1972, pp 61-64.

WULF, W. and BELL, C.G:

"Cmmp a Multi-Mini-Processor", AFIPS Fall Joint Computer Conf., American AFIPS Proc., Vol. 22, Washington, Spartan, DC, 1962, pp 97-107. 1972 FJCC, pp 765-777.

APPENDICES

Appendix A

SELECTED PARALLEL PROGRAMS

```

1:      c      *** Program 4.1 ***
2:      c
3:      c      Parallel sequential search algorithm
4:      c      version 1.0 of an unordered set of
5:      c      real numbers.
6:      c
7:      c      k      : an unordered array. real values.
8:      c      key    : the key value to be searched.
9:      c      n      : the problem size.
10:     c      nproc   : Number of paths.
11:     c      nelem    : number of element per sub-group.
12:     c
13:     c      this algorithm assumes the follolwing assumptions.
14:     c
15:     c      1 : keys in the array are not repeated.
16:     c      2 : unsuccessful searches are not considered
17:     c      key is the ikey.th element of k.
18:     c      3 : processors broadcast their seaching result
19:     c      through a shared memory variable "found"
20:     c      which takes 0 for false and 1 for true.
21:     c
22:     c      dimension k(8192),itime(144),ind(100),t(100)
23:     c      integer fi,found,fausse,vraie,find
24:     c      declare the shared data.
25:     c
26:     c      $shared n,m,k,ind,find,found,fausse,vraie,ikey,fi,itime
27:     c      $shared nproc,key
28:     c      $region cr
29:     c      initialize parallelism .
30:     c
31:     c      $usepar
32:     c      n=8192
33:     c      fausse=0
34:     c      vraie= 1
35:     c      obtain the number of active processors.
36:     c      call nprocs( nproc )
37:     c      random generation of the array k.
38:     c      size fixed to 8192 elements.
39:     c      iran=ran(-1)
40:     c      do 100 i=1,n
41:     100 k(i)=n-i
42:     c
43:     c      do 105 i=1,100
44:     c      x=ran(1)
45:     c      ind(i)=1+(1.0-x)*n
46:     105 continue
47:     c      100 locations where the target keys are
48:     c      supposed to be found in are randomly generated
49:     c
50:     c      sort ind in increasing order of values.
51:     c      in=100
52:     106 if(in.eq.1) goto 107
53:     c      max=1
54:     c      do 108 i=1,in
55:     c      if(ind(max).lt.ind(i)) max=i
56:     108 continue
57:     c      sav=ind(in)
58:     c      ind(in)=ind(max)
59:     c      ind(max)=sav
60:     c      in=in-1
61:     c      goto 106
62:     107 continue
63:     c
64:     c      do 120 irep=1,100
65:     c      ikey=ind(irep)
66:     c      key=k(ikey)

```

```

67:      c
68:      c
69:      c
70:      $doall 200
71:          call timest
72:      200  $parend
73:      c
74:      c
75:      found=fausse
76:      fi=0
77:      $dopar 300 ipath=1,nproc
78:          c      check if the target key is not found yet by
79:          c      other processor than me.
80:          if(found.eq.vraie) goto 330
81:          c      the flag is tested every key comparison.
82:          nelem=n/nproc
83:          is=(ipath-1)*nelem+1
84:          ie=ipath*nelem
85:          if(ipath.eq.nproc) ie=n
86:          j=is
87:      310  if(j.gt.ie.or(found.eq.vraie) goto 320
88:          if(key.ne.k(j)) goto 322
89:          found=vraie
90:          fi=j
91:          goto 320
92:      322  j=j+1
93:          goto 310
94:      320  continue
95:      330  continue
96:      300  $parend
97:      find=fi
98:      c
99:      c
100:     c      stop the timing
101:     $doall 400
102:         call timeout(itime)
103:     400  $parend
104:
105:     t(irep)=0
106:     do 450 i=1,nproc
107:         is=24*(i-1)
108:         tim=itime(is+1)+.001*itime(is+2)
109:         if( t(irep).lt.tim) t(irep)=tim
110:     450  continue
111:     120  continue
112:     write(*,20)t
113:     $stop
114:     20  format(10f7.2)
115:     $end

```

```

1:      c      *** Program 4.2 ***
2:      c
3:      c      Parallel sequential search algorithm
4:      c      Version 2.0 of an unordered set of
5:      c      real numbers.
6:      c
7:      c      k      : an unordered array. real values.
8:      c      key    : the key value to be searched.
9:      c      n      : the problem size.
10:     c      nproc   : number of sub-groups.
11:     c      nelem    : number of element per sub-group.
12:     c
13:     c      This algorithm assumes the follolwing assumptions.
14:     c
15:     c      1 : keys in the array are not repeated.
16:     c      2 : unsuccessful searches are not considered
17:     c          key is the ikey.th element of k.
18:     c      3 : processors broadcast their seaching result
19:     c          through a shared memory variable "found"
20:     c          which takes 0 for false and 1 for true.
21:     c
22:     c      dimension k(1024*1024),itime(144),ind(100),t(100)
23:     c      integer fi,found,fausse,vraie,find
24:     c          declare the shared data.
25:     c
26:     c      $shared n,m,k,ind,find,found,fausse,vraie,ikey,fi,itime
27:     c      $shared nproc,key
28:     c      $region cr
29:     c          initialize parallelism .
30:     c
31:     c      $usepar
32:     c          n=1024*1024
33:     c          fausse=0
34:     c          vraie= 1
35:     c          obtain the number of active processors.
36:     c      call nprocs( nproc )
37:     c          random generation of the array k.
38:     c          size fixed to 1 Million elements.
39:     c      iran=ran(-1)
40:     c      do 100 i=1,n
41: 100    c      k(i)=n-i
42:     c
43:     c      do 105 i=1,100
44:     c          x=ran(1)
45:     c          ind(i)=1+(1.0-x)*n
46: 105    c      continue
47:     c          100 locations where the target keys are
48:     c          supposed to be found in are randomly generated
49:     c
50:     c          sort ind in increasing order of values.
51:     c      in=100
52: 106    c      if(in.eq.1) goto 107
53:     c          max=1
54:     c          do 108 i=1,in
55:     c              if(ind(max).lt.ind(i)) max=i
56: 108    c      continue
57:     c          sav=ind(in)
58:     c          ind(in)=ind(max)
59:     c          ind(max)=sav
60:     c          in=in-1
61:     c          goto 106
62: 107    c      continue
63:     c
64:     c      do 120 irep=1,100
65:     c          ikey=ind(irep)
66:     c          key=k(ikey)

```

```

67:      c
68:      c                      start timing procedure.
69:      c
70:      $doall 200
71:          call timest
72:      200      $parend
73:      c
74:      c
75:      found=fausse
76:      fi=0
77:      $dopar 300 ipath=1,nproc
78:          c          check if the target key is not found yet by
79:          c          other processor than me.
80:          if(found.eq.vraie) goto 330
81:          c          the flag is tested every key comparison.
82:          j=ipath
83:      310      if(j.gt.n.or(found.eq.vraie) goto 320
84:          if(key.ne.k(j)) goto 322
85:          found=vraie
86:          fi=j
87:          goto 320
88:      322      j=j+nproc
89:          goto 310
90:      320      continue
91:      330      continue
92:      300      $parend
93:      find=fi
94:      c
95:      c                      stop the timing
96:      c
97:      $doall 400
98:          call timeout(itime)
99:      400      $parend
100:
101:      t(irep)=0
102:      do 450 i=1,nproc
103:          is=24*(i-1)
104:          tim=itime(is+1)+.001*itime(is+2)
105:          if( t(irep).lt.tim) t(irep)=tim
106:      450      continue
107:      120      continue
108:      write(*,20)t
109:      $stop
110:      20      format(10f7.2)
111:      $end

```

```

1:      c
2:      c      *** Program 4.3 ***
3:      c
4:      c      Parallel binary search algorithm
5:      c      version 1.0.
6:      c      the number of paths is always equal
7:      c      to the number of activated processors
8:      c
9:      c
10:     c      k      non-decreasing ordered array of size n.
11:     c      key    key to be searched in k.
12:     c
13:     c      nb : all these variables are shared.
14:     c
15:     dimension k(8*1024),itime(144)
16:     $shared   k,n,fi,itime,key,ikey,np
17:     $shared   nproc
18:     $region   cr
19:     integer   low,mid,high,fi,find
20:     c
21:     c      start parallelism.
22:     c
23:     $usepar
24:     n=8*1024
25:     c      compute the number of activated processors.
26:     call nprocs( nproc)
27:     c      the array elements are randomly generated.
28:     iran=ran(-1)
29:     do 90 i=1,n
30:     90      k(i)=i
31:     c      start timing.
32:     $doall 150
33:         call timest
34:     150 $parend
35:     c
36:     c      n different target keys are searched in
37:     c      every iteration.
38:     c
39:     $dopar 200 me=1,nproc
40:         ln=n
41:         do 140 irep=me,ln,nproc
42:             x=ran(1)
43:             iky=1+(1.0-x)*n
44:             ky=k(iky)
45:             lfi=0
46:             c      perform the binary search
47:                 low=1
48:                 high=n
49:             300         if(high-low.eq.0) goto 310
50:                         mid=(low+high)/2
51:                         if(ky-k(mid)) 320,321,322
52:             320         if(ky.lt.k(low)) goto 323
53:                         high=mid-1
54:                         goto 300
55:             321         high=mid
56:                         low=mid
57:                         goto 300
58:             322         if(ky.gt.k(high)) goto 323
59:                         low=mid+1
60:                         goto 300
61:             310         continue
62:             if(ky.eq.k(high)) lfi=high
63:             323         continue
64:             140         continue
65:             200         $parend
66:             c      stop timing.

```

```
67:      $doall 500
68:          call timeout(itime)
69:      500    $parend
70:      c      print the time .
71:          write(*,10)
72:      call printt(itime)
73:      $stop
74:      10     format(/,'      BS version 1.0 ',/)
75:      $end
```

```

1:      c
2:      c      *** Program 4.4 ***
3:      c
4:      c      Parallel binary search algorithm
5:      c      version 2.0 .
6:      c      the number of paths is always
7:      c      equal to the number of activated
8:      c      processors.
9:      c
10:     c
11:     c      The way the array is divided is as follows
12:     c      if p is the number of processor and if me
13:     c      is the processor-name the me has access to
14:     c      the locations indexed by p*k+me+1 where k
15:     c      is an integer.
16:     c
17:     c      k      non-decreasing ordered array of size n.
18:     c      key    key to be searched in k.
19:     c
20:     c      nb : all these variables are shared.
21:     c
22:     dimension k(8*1024),itime(144)
23:     $shared   k,n,fi,itime,key,ikey
24:     $shared   nproc
25:     $region   cr
26:     integer   low,mid,high,fi,find,rak
27:     c
28:     c      start parallelism.
29:     c
30:     $usepar
31:     n=8*1024
32:     c      compute the number of activated processors.
33:     call nprocs( nproc)
34:     c      the array elements are randomly fenerated.
35:     iran=ran(-1)
36:     do 90 i=1,n
37:     90      k(i)=i+ran(1)
38:     c      start timing.
39:     $doall 150
40:         call timest
41:     150 $parend
42:     c
43:     c      a 1000 different target keys are searched in
44:     c      every iteration.
45:     c
46:     $dopar 200 me=1,nproc
47:     do 140 irep=me,1000,nproc
48:         x=ran(1)
49:         ikey=1+(1.0-x)*n
50:         ky=k(iky)
51:         fi=0
52:         np=nproc
53:         kar=key
54:     c      perform the binary search on this path.
55:         low=me
56:         high=n+low-np
57:     300     if(high-low.eq.0.or.fi.ne.0) goto 310
58:         mid=np*((low-me)/np+(high-me)/np)/2+me
59:         if(kar-k(mid)) 320,321,322
60:     320         high=mid-np
61:         goto 300
62:     321         high=mid
63:         low=mid
64:         goto 300
65:     322         low=mid+np
66:         goto 300

```

```
67:      310      continue
68:      if(kar.eq.k(high)) fi=high
69:      200      $parend
70:      find=fi
71:      140      continue
72:      c              stop timing.
73:      $doall 500
74:      call timout(itime)
75:      500      $parend
76:      c              print the time .
77:      call printt(itime)
78:      $stop
79:      $end
```

```

1:      c      *** Program 4.5 ***
2:      c
3:      c      Parallel binary search algorithm
4:      c      version 3.0.
5:      c      The number of parallel paths is always
6:      c      equal to the number of activated
7:      c      processors.
8:      c
9:      c      The original array is divided into p sub-arrays.
10:     c      Each processor takes one sub-array of length n/p,
11:     c      compares the key with the contents of the middle
12:     c      point location of the associated sub-array. If
13:     c      the comparison is successful, the algorithm
14:     c      terminates , otherwise a decision is made to
15:     c      which of the p sub-arrays does contain the
16:     c      target key. the process is repeated with the
17:     c      new selected and smaller sub-array. Thus the
18:     c      array gets smaller after each iteration by a
19:     c      factor of 2*nproc.
20:     c
21:     c      we assume the search to be successful and
22:     c      therefor the present algorithm does not
23:     c      consider the unsuccessful case.
24:     c
25:     c      k      non-decreasing ordered array of size n.
26:     c      key    key to be searched in k.
27:     c
28:     c      nb : all these variables are shared.
29:     c
30:     dimension k(8*1024),itime(144)
31:     $shared   k,n,fi,itime,key,ikey,nproc,slow,shigh,sl,sh
32:     $region   cr
33:     integer   slow,shigh,low,mid,high,sl,sh,fi,find
34:     c
35:     c      start parallelism.
36:     c
37:     $usepar
38:     n=8*1024
39:     c      compute the number of activated processors.
40:     call nprocs( nproc)
41:     c      the array elements are randomly fenerated.
42:     iran=ran(-1)
43:     do 90 i=1,n
44:     90      k(i)=i+ran(1)
45:     c      start timing.
46:     $doall 150
47:         call timest
48:     150 $parend
49:     c
50:     c      a 100 different target keys are searched in
51:     c      every iteration.
52:     do 140 irep=1,100
53:         x=ran(1)
54:         ikey=1+(1.0-x)*n
55:         key=k(ikey)
56:         fi=0
57:     c      start processing.
58:     slow=1
59:     shigh=n
60:     c
61:     250 if(shigh-slow.eq.0) goto 260
62:     $dopar 200 me=1,nproc
63:         slow=(shigh+1-slow)/nproc
64:         low=(me-1)*nelem+slow
65:         high=low+nelem-1
66:         if(me.eq.nproc) high=shigh

```

```

67:      c                      compute the middle point
68:      mid=(low+high)/2
69:      if(key.ne.k(mid))goto 310
70:      c                      the searched key is found at mid
71:      sl=mid
72:      sh=mid
73:      goto 330
74:      310      if(k(low).gt.key.or.key.gt.k(mid)) goto 320
75:      c                      the key is located in the interval
76:      c                      [k(low)...k(mid)]
77:      sh=mid-1
78:      sl=low
79:      goto 330
80:      320      if(k(mid).gt.key.or.key.gt.k(high))goto 330
81:      c                      the key is located in the interval
82:      c                      ]k(mid)...k(high)]
83:      sl=mid+1
84:      sh=high
85:      330      continue
86:      200      $parend
87:      slow=sl
88:      shigh=sh
89:      goto 250
90:      260      continue
91:      140      continue
92:      c                      stop timing.
93:      $doall 500
94:      call timeout(itime)
95:      500      $parend
96:      c                      print the time .
97:      call printt(itime)
98:      $stop
99:      $end

```

```

1:      c
2:      c      *** Program 4.6 ***
3:      c
4:      c      Fast sequential jump searching algorithm :
5:      c      ****
6:      c
7:      c      simple jump searching algorithm
8:      c
9:      c      the jump size is defined to be square root of the
10:     c      size of the array to be searched.
11:     c
12:     c      k      non-decreasing ordered array of size n.
13:     c      key    key to be searched in k.
14:     c
15:     c      nb : all these variables are shared.
16:     c
17:     dimension k(64*1024),itime(144),ind(100)
18:     $shared   k,n,fi,itime,key,ikey,nproc,jsizl
19:     $region   cr
20:     integer   fi
21:     c
22:     c      start parallelism.
23:     c
24:     $usepar
25:     n=64*1024
26:     jsizl=n**(1./2.)
27:     write(*,10)
28:     10      format(/,'          SJS algorithm ',/)
29:     c      compute the number of activated processors.
30:     nproc=0
31:     $doall 125
32:         $enter cr
33:         me=nproc
34:         nproc=nproc+1
35:         $exit  cr
36:     125 $parend
37:     c      the array elements are randomly fenerated.
38:     id=ran(-1)
39:     do 130 i=1,n
40:     130     k(i)=i
41:     c      a N locations where the target keys are
42:     c      supposed to be found in are randomly generated
43:     c
44:     c
45:     do 135 i=1,100
46:         x=ran(1)
47:         ind(i)=1+(1.0-x)*n
48:     135 continue
49:     c      100 locations where the target keys are
50:     c      supposed to be found in are randomly generated
51:     c
52:     do 140 irep=1,100,4
53:         ikey=ind(irep)
54:         key=k(ikey)
55:     c
56:     c      start timing procedure.
57:     c
58:     $doall 145
59:         call timest
60:     145 $parend
61:     c
62:     do 155 iii=1,1000
63:     $doall 150
64:         ky=key
65:         lfi=0
66:     c

```

```

67:      c                               simple jump searching method. 398
68:      c
69:      i=me*jsizl+1
70:      jsiz=nproc*jsizl
71:      160  if(k(i).ge.ky.or.i.gt.n)goto 165
72:          i=i+jsiz
73:          goto 160
74:      165  continue
75:          if (i.le.n) goto 170
76:          ii=1
77:      180  if( ii.gt.nproc.or.i.le.n) goto 185
78:          i=i-jsizl
79:          ii=ii+1
80:          goto 180
81:      185  continue
82:          i=i+jsizl
83:      170  continue
84:      c      Find the block where the key is likely to be in
85:      c      For nproc processors there are nproc sub-blocks to choos
86:      c      from. The selected sub-block will be also searched in
87:      c      parallel.
88:          ii=1
89:      190  if ( ii.gt.nproc ) goto 195
90:          lasti=i-jsizl
91:      c      Ensure that lasti is defined.
92:          if(lasti.gt.0) goto 200
93:          ii=nproc+1
94:          lasti=me+1
95:          goto 190
96:      200  continue
97:          i=lasti
98:          if( k(i).gt.ky) goto 205
99:          ii=nproc+1
100:         i=lasti+me
101:      205  continue
102:          ii=ii+1
103:          goto 190
104:      195  continue
105:      c      forward sequential search of the just passed block
106:      210  if(k(i).ge.ky)goto 215
107:          i=i+nproc
108:          goto 210
109:      215  continue
110:          if(i.le.n.and.k(i).eq.ky)lfi=i
111:      c      target key is found at location i
112:      150  $parend
113:      155  continue
114:      c      stop timing.
115:      $doall 220
116:          call timeout(itime)
117:      220  $parend
118:      c      print the time .
119:          tim=itime(1)+itime(2)*.001
120:          write(*,30)tim
121:      30  format(f8.3)
122:      140  continue
123:      $stop
124:      $end

```

```

1:      c
2:      c      *** Program 4.7 ***
3:      c
4:      c      Fast sequential jump searching algorithm :
5:      c      *****
6:      c
7:      c      two-level simple jump searching algorithm
8:      c
9:      c      two levels of jumps are involved. the first jump
10:     c      is defined by  $n^{1/2}$  and the second level jump
11:     c      size is  $n^{1/4}$ . n is the array size.
12:     c
13:     c      the sequential part is carried out forwards.
14:     c
15:     c      k      non-decreasing ordered array of size n.
16:     c      key    key to be searched in k.
17:     c
18:     c      nb : all these variables are shared.
19:     c
20:     c      dimension k(64*1024),itime(144),ind(100)
21:     c      $shared k,n,fi,itime,key,ikey,nproc,jsiz1,jsiz2
22:     c      $region cr
23:     c      integer fi
24:     c
25:     c      start parallelism.
26:     c
27:     c      $usepar
28:     c      write(*,10)
29:     10  c      format(/,'      TLSJS version 1.0',/)
30:     c      n=64*1024
31:     c      jsiz1=n**(1./2.)
32:     c      jsiz2=n**(1./4.)
33:     c      compute the number of activated processors.
34:     c      nproc=0
35:     c      $doall 100
36:     c      $enter cr
37:     c      me=nproc
38:     c      nproc=nproc+1
39:     c      $exit cr
40:     100 c      $parend
41:     c      the array elements are randomly fenerated.
42:     c      iran=ran(-1)
43:     c      do 110 i=1,n
44:     110 c      k(i)=i
45:     c      100 locations where the target key is to be found
46:     c      in are randomly generated.
47:     c
48:     c      do 115 irep=1,100
49:     c      x=ran(1)
50:     c      ind(irep)=1+(1.0-x)*n
51:     115 c      continue
52:     c      do 120 irep=1,25
53:     c      ikey=ind(irep)
54:     c      key = k(ikey)
55:     c      start timing.
56:     c      $doall 125
57:     c      call timest
58:     125 c      $parend
59:     c
60:     c      a N locations where the target keys are
61:     c      supposed to be found in are randomly generated
62:     c
63:     c
64:     c      $doall 130
65:     c      do 135 iii=1,1000
66:     c      ky=key

```

```

67:          lfi=0
68:      c
69:          two-level simple jump searching method.
70:      c
71:          i=me*jsiz1+1
72:          jsiz=nproc*jsiz1
73:      140      if(k(i).ge.ky.or.i.gt.n)goto 145
74:              i=i+jsiz
75:              goto 140
76:      145      continue
77:      c          Adjust the index if it is greater than n.
78:          if (i.le.n) goto 150
79:              ii=1
80:      155          if( ii.gt.nproc.or.ii.le.n) goto 160
81:                  i=i-jsiz1
82:                  ii=ii+1
83:                  goto 155
84:      160          continue
85:                  i=i+jsiz1
86:      150      continue
87:      c          a possible target key was just passed.
88:      c          apply the second level of the method.
89:          ii=1
90:      165          if( ii.gt.nproc) goto 170
91:          lasti=i-jsiz1
92:      c          Ensure that lasti is defined.
93:          if(lasti.gt.0) goto 175
94:              ii=nproc + 1
95:              lasti=me*jsiz2+1
96:              goto 165
97:      175      continue
98:              i=lasti
99:              if( k(i).gt.ky) goto 180
100:                  ii=nproc+1
101:                  i=lasti+me*jsiz2
102:      180      continue
103:                  ii=ii+1
104:                  goto 165
105:      170      continue
106:      c          Second block search level
107:          jsiz=nproc*jsiz2
108:      185      if(k(i).ge.ky.or.i.gt.n)goto 190
109:              i=i+jsiz
110:              goto 185
111:      190      continue
112:      c          Adjust once more the index i
113:          if (i.le.n) goto 200
114:              ii=1
115:      205          if( ii.gt.nproc.or.ii.le.n) goto 210
116:                  i=i-jsiz2
117:                  ii=ii+1
118:                  goto 205
119:      210          continue
120:                  i=i+jsiz2
121:      200      continue
122:
123:      c          a possible target key was just passed.
124:          ii=1
125:      215          if( ii.gt.nproc) goto 220
126:          lasti=i-jsiz2
127:      c          Ensure that lasti is defined.
128:          if(lasti.gt.0) goto 225
129:              ii=nproc + 1
130:              lasti=me+1
131:              goto 215
132:      225      continue

```

```

133:      i=lasti
134:      if( k(i).gt.ky) goto 230
135:          ii=nproc+1
136:          i=lasti+me
137: 230      continue
138:          ii=ii+1
139:          goto 215
140: 220      continue
141:  c
142:  c          forward sequential search of the just passed
143:  c          block.
144: 235      if(k(i).ge.ky)goto 240
145:          i=i+nproc
146:          goto 235
147: 240      continue
148:          if(i.le.n.and.k(i).eq.ky)lfi=i
149:  c          target key is found at location i
150: 135      continue
151: 130      $parend
152:  c          stop timing.
153:      $doall 500
154:          call timeout(itime)
155: 500      $parend
156:  c          print the time .
157:          tim1=itime(1)+.001*itime(2)
158:          write(*,20)tim1
159: 20      format(2f8.3)
160: 120      continue
161:      $stop
162:      $end

```

```

1:      c
2:      c      *** Program 4.8 ***
3:      c
4:      c      Fast sequential jump searching algorithm :
5:      c      ****
6:      c
7:      c      two-level fixed jump searching algorithm
8:      c
9:      c      two levels of jumps are involved. the first jump
10:     c      is defined by  $n^{2/3}$  and the second level jump
11:     c      size is  $n^{1/3}$ . n is the array size.
12:     c
13:     c      the sequential part is carried out forwards.
14:     c
15:     c      k      non-decreasing ordered array of size n.
16:     c      key    key to be searched in k.
17:     c
18:     c      nb : all these variables are shared.
19:     c
20:     dimension k(64*1024),itime(144),ind(100)
21:     $shared   k,n,fi,itime,key,ikey,nproc,jsiz1,jsiz2
22:     $region   cr
23:     integer   fi
24:     c
25:     c      start parallelism.
26:     c
27:     $usepar
28:     write(*,10)
29:     10      format(/,'      TLFJS version 1.0',/)
30:     n=64*1024
31:     jsiz1=n**(2./3.)
32:     jsiz2=n**(1./3.)
33:     c      compute the number of activated processors.
34:     nproc=0
35:     $doall 100
36:     $enter cr
37:     me=nproc
38:     nproc=nproc+1
39:     $exit cr
40:     100     $parend
41:     c      the array elements are randomly fenerated.
42:     iran=ran(-1)
43:     do 110 i=1,n
44:     110     k(i)=i
45:     c      100 locations where the target key is to be found
46:     c      in are randomly generated.
47:     c
48:     do 115 irep=1,100
49:     x=ran(1)
50:     ind(irep)=1+(1.0-x)*n
51:     115     continue
52:     do 120 irep=1,100
53:     ikey=ind(irep)
54:     key = k(ikey)
55:     c      start timing.
56:     $doall 125
57:     call timest
58:     125     $parend
59:
60:     c      a N locations where the target keys are
61:     c      to be found in are randomly generated
62:     c
63:     c
64:     $doall 130
65:     do 135 iii=1,1000
66:     ky=key

```

```

67:          lfi=0
68:      c
69:      c          two-level fixed jump searching method.
70:      c
71:          i=me*jsiz1
72:          if(me.eq.0) i=1
73:          jsiz=nproc*jsiz1
74:      140      if(k(i).ge.ky.or.i.gt.n)goto 145
75:              i=i+jsiz
76:              goto 140
77:      145      continue
78:      c          Adjust the index if it is greater than n.
79:          if (i.le.n) goto 150
80:              ii=1
81:      155      if( ii.gt.nproc.or.ii.le.n) goto 160
82:              i=i-jsiz1
83:              ii=ii+1
84:              goto 155
85:      160      continue
86:              i=i+jsiz1
87:      150      continue
88:      c          a possible target key was just passed.
89:      c          apply the second level of the method.
90:          ii=1
91:      165      if( ii.gt.nproc) goto 170
92:          lasti=i-jsiz1
93:      c          Ensure that lasti is defined.
94:          if(lasti.gt.0) goto 175
95:              ii=nproc + 1
96:              lasti=me*jsiz2
97:              if(me.eq.0) lasti=1
98:              goto 165
99:      175      continue
100:          i=lasti
101:          if( k(i).gt.ky) goto 180
102:              ii=nproc+1
103:              i=lasti+me*jsiz2
104:      180      continue
105:              ii=ii+1
106:              goto 165
107:      170      continue
108:      c          Second block search level
109:          jsiz=nproc*jsiz2
110:      185      if(k(i).ge.ky.or.i.gt.n)goto 190
111:              i=i+jsiz
112:              goto 185
113:      190      continue
114:      c          Adjust once more the index i
115:          if (i.le.n) goto 200
116:              ii=1
117:      205      if( ii.gt.nproc.or.ii.le.n) goto 210
118:              i=i-jsiz2
119:              ii=ii+1
120:              goto 205
121:      210      continue
122:              i=i+jsiz2
123:      200      continue
124:
125:      c          a possible target key was just passed.
126:          ii=1
127:      215      if( ii.gt.nproc) goto 220
128:          lasti=i-jsiz2
129:      c          Ensure that lasti is defined.
130:          if(lasti.gt.0) goto 225
131:              ii=nproc + 1
132:              lasti=me+1

```

```

133:      goto 215
134: 225    continue
135:      i=lasti
136:      if( k(i).gt.ky) goto 230
137:      ii=nproc+1
138:      i=lasti+me
139: 230    continue
140:      ii=ii+1
141:      goto 215
142: 220    continue
143:  c
144:  c      forward sequential search of the just passed
145:  c      block.
146: 235    if(k(i).ge.ky)goto 240
147:      i=i+nproc
148:      goto 235
149: 240    continue
150:      if(i.le.n.and.k(i).eq.ky)lfi=i
151:  c      target key is found at location i
152: 135    continue
153: 130    $parend
154:  c      stop timing.
155:      $doall 500
156:      call timeout(itime)
157: 500    $parend
158:  c      print the time .
159:      tim=itime(1)+.001*itime(2)
160:      write(*,20)tim
161: 20     format(f8.3)
162: 120    continue
163:      $stop
164:      $end

```

```

1:      c
2:      c          *** Program 5.1 ***
3:      c
4:      c $CHAREQU
5:      c          psim : parallel simple string matching alg.
6:      c
7:      c          this is the simple and obvious way to search
8:      c          a string of characters for all occurrences
9:      c          of a string pat in another string . the search
10:     c          is started by matching characters from the left-
11:     c          most character of both strings.
12:     c
13:     dimension ind(100),itime(144),t(10)
14:     character*1 string(500000),pat(15),line(125)
15:     $shared itime,strlen,patlen,nproc,ikey,string
16:     integer strlen,strmax,patlen,ptl
17:     $usepar
18:     c          input the string of characters
19:     strlen=1
20:     strmax=500000
21:     do 50 iline=1,4000
22:         read(*,10)line
23:         do 50 i=1,125
24:             string(strlen)=line(i)
25:             strlen=strlen+1
26:         if(strlen.gt.strmax) strlen=strmax
27:     c          find number of processors .
28:     call nprocs(nproc)
29:     nelelem=strlen/nproc
30:     c          start random generator
31:     iran=ran(-1)
32:     do 400 patlen = 6,15
33:         c          generate a 100 random positions in string
34:         do 405 i=1,100
35:             x=ran(1)
36:             ind(i)=1+(1.0-x)*(strlen+1-patlen)
37:         c
38:         do 410 irep=1,100,10
39:             ikey=ind(irep)
40:         c          start timing.
41:         $doall 425
42:             call timest
43:         425 $parend
44:         c          start parallel processing.
45:         $dopar 420 ipath=1,nproc
46:             nelelem=strlen/nproc
47:             is=(ipath-1)*nelelem+1
48:             ie=ipath*nelelem+patlen-1
49:             if(ipath.eq.nproc)ie=strlen
50:             ptl=patlen
51:             iky=ikey
52:         c          copy pat from string starting from
53:         c          position ikey.
54:         do 415 i=1,ptl
55:             ik=iky+i-1
56:             pat(i)=string(ik)
57:         c
58:         c
59:         c          the simple string matching algorithm.
60:         i=is
61:         300 if(i.gt.ie+1-ptl) goto 350
62:         j=1
63:         k=i
64:         320 if(string(k).ne.pat(j)) goto 340
65:         if(j.ne.ptl)goto 330
66:         c          an occurrence of pat is found.

```

```
67:          i=i+ptl
68:          goto 300
69:      330      k=k+1
70:          j=j+1
71:          goto 320
72:      340      i=i+1
73:          goto 300
74:      350      continue
75:      420      $parend
76:      c
77:      c          stop timing.
78:          $doall 435
79:              call timeout(itime)
80:      435      $parend
81:      c          compute timing in second.
82:          it=irep/10+1
83:          t(it)=itime(3)+.001*itime(4)
84:          if(it.eq.10)write(*,20)t
85:      410      continue
86:      400      continue
87:          $stop
88:      10      format(125a1)
89:      20      format(10f7.2)
90:          $end
```

```

1:      c
2:      c      *** Program 5.2 ***
3:      c
4:      c      $CHAREQU
5:      c      prk : parallel Rabin-Karp string matching alg.
6:      c
7:      c      This is the Parallel Rabin-Karp string searching
8:      c      algorithm . It is based on a hashing function H.
9:      c      Pat and a string from the text are hashed, h1 and h2
10:     c      represent their respective hash values. they are equal
11:     c      if and only if h1 equal to h2.
12:     c
13:     c      dimension ind(100),itime(144),t(10)
14:     c      character*1 string(500000),pat(15),line(125)
15:     c      $shared itime,strlen,patlen,nproc,ikey,string
16:     c      integer strlen,strmax,patlen,ptl
17:     c      integer q,d,dm,h1,h2
18:     c      $usepar
19:     c      input the string of characters
20:     c      strlen=1
21:     c      strmax=500000
22:     c      do 50 iline=1,4000
23:     c          read(*,10)line
24:     c          do 50 i=1,125
25:     c              string(strlen)=line(i)
26:     c              strlen=strlen+1
27:     c          if(strlen.gt.strmax) strlen=strmax
28:     c          find number of processors .
29:     c      call nprocs(nproc)
30:     c      start random generator
31:     c      iran=ran(-1)
32:     c      do 400 patlen = 6,15
33:     c          generate a 100 random positions in string
34:     c          do 405 i=1,100
35:     c              x=ran(1)
36:     c              ind(i)=1+(1.0-x)*(strlen+1-patlen)
37:     c          do 410 irep=1,100,10
38:     c              ikey=ind(irep)
39:     c          start timing.
40:     c      $doall 425
41:     c          call timest
42:     c      $parend
43:     c      start parallel processing.
44:     c      $dopar 420 ipath=1,nproc
45:     c          nelem=strlen/nproc
46:     c          is=(ipath-1)*nelem+1
47:     c          ie=ipath*nelem+patlen-1
48:     c          if(ipath.eq.nproc)ie=strlen
49:     c          iky=ikey
50:     c          copy pat from string starting from
51:     c          position ikey.
52:     c          do 415 i=1,ptl
53:     c              ik=iky+i-1
54:     c              pat(i)=string(ik)
55:     c          415
56:     c
57:     c
58:     c      the RK string matching algorithm.
59:     c      Initializations
60:     c      q=33554393
61:     c      d=128
62:     c      m=patlen
63:     c      dm=1
64:     c      do 200 i=1,m-1
65:     c          dm=mod(d*dm,q)
66:     c      h1=0

```

```

67:      do 210 i=1,m
68: 210    h1=mod( h1*d + ichar(pat(i)),q)
69:      h2=0
70:      do 220 i=is,is+m-1
71: 220    h2=mod( h2*d + ichar(string(i)),q)
72:
73: 300    if(i.gt.ie+1-ptl) goto 350
74:          if( h1.ne.h2) goto 310
75:  c          an occurrence of pat is found
76:          fi = i-m
77: 310      h2=mod( h2 + d*q - ichar(string(i)),q)
78:          h2=mod( h2*d + ichar( string(i+m)),q)
79:          i=i+1
80:          goto 300
81: 350      continue
82: 420      $parend
83:  c
84:  c          stop timing.
85:          $doall 435
86:          call timeout(itime)
87: 435      $parend
88:  c          compute timing in second.
89:          it=irep/10+1
90:          t(it)=itime(3)+.001*itime(4)
91:          if(it.eq.10)write(*,20)t
92: 410      continue
93: 400      continue
94:      $stop
95: 10      format(125a1)
96: 20      format(11f7.2)
97:      $end

```

```

1:      c
2:      c      *** Program 5.3 ***
3:      c
4:      $CHAREQU
5:      c      the parallel version of the knuth, morris and
6:      c      pratt ' s string matching algorithm.
7:
8:      c      the idea of this method is similar to the
9:      c      sim algorithm except that a precomputed table
10:     c      is used to jump through part of the string
11:     c      that could not be the searched pattern.
12:     c      for more details how the next table is computed
13:     c      see knuth, morris and pratt's paper.
14:     c
15:     character*1 string(500000),line(125),c,pat(16),lpat(16),
+achar,arau
16:     dimension itime(144),t(10),next(16),lnext(16),ind(100)
17:     $shared string,pat,next,itime,ind,strlen,patlen,nproc,
+ikey ,arau
18:     integer patlen,strlen,strmax,ptl
19:     $usepar
20:     c      arau is a character not present in the string.
21:     arau=char(0)
22:     c      input the string .
23:     strmax=500000
24:     strlen=1
25:     do 50 iline=1,4000
26:         read(*,10)line
27:         do 50 i=1,125
28:             string(strlen)=line(i)
29:             strlen=strlen+1
30:         if( strlen .gt. strmax) strlen=strmax
31:         c      find number of processors.
32:     call nprocs( nproc)
33:     c
34:     c      start random generator
35:     iran=ran(-1)
36:     do 400 patlen = 6,15
37:         c      generate 100 random locations in string
38:         do 405 i=1,100
39:             x=ran(1)
40:             ind(i)=1+(1.0-x)*(strlen+1-patlen)
41:         c
42:         do 410 irep=1,100,10
43:             ikey=ind(irep)
44:             c      copy pat from string starting from
45:             c      position ikey.
46:             do 415 i=1,patlen
47:                 ik=ikey+i-1
48:                 pat(i)=string(ik)
49:             c      start timing.
50:             $doall 420
51:             call timest
52:             420 $parend
53:         c
54:         c
55:         c      next setting up algorithm.
56:         c
57:         c
58:         j=1
59:         k=0
60:         next(1)=0
61:         100 if(j.ge.patlen) goto 102
62:         c      k=f(k)
63:         103 if(k.le.0.or.pat(j).eq.pat(k)) goto 104
64:         k=next(k)

```

```

65:          goto 103
66: 104      j=j+1
67:          k=k+1
68:          next(j)=k
69:          if(pat(j).eq.pat(k))next(j)=next(k)
70:          goto 100
71: 102      continue
72:  c          start parallelism.
73:          $dopar 425 ipath=1,nproc
74:          nelem=strlen/nproc
75:          is=(ipath-1)*nelem+1
76:          ie=ipath *nelem+patlen-1
77:          if(ipath.eq.nproc)ie=strlen
78:          ptl=patlen
79:  c          copy lpat from pat & lnext from next
80:          do 305 i=1,ptl
81:              lnext(i)=next(i)
82: 305          lpat(i)=pat(i)
83:  c
84:  c          the knuth, morris and pratt string
85:  c          matching algorithm.
86:  c
87:          achar=lpat(1)
88:          lpat(ptl+1)=arau
89:          lnext(ptl+1)=-1
90:          j=1
91:          k=is
92:  c          get started. j = 1
93: 300      if(k.gt.ie.or.string(k).eq.achar)goto 310
94:          k=k+1
95:          if(k.gt.ie)goto 350
96:          goto 300
97: 310      continue
98:  c          char matched
99: 315      j=j+1
100:          k=k+1
101:          if(k.gt.ie)goto 350
102:  c          loop
103: 320      if(string(k).eq.lpat(j))goto 315
104:          j=lnext(j)
105:          if(j.eq.1)goto 300
106:          if(j.ne.0)goto 325
107:          j=1
108:          k=k+1
109:          if(k.gt.ie)goto 350
110:          goto 300
111: 325      if(j.gt.0) goto 320
112:  c          an occurrence of pat is found.
113:          j=1
114:          goto 300
115: 350      continue
116:  c          the algorithm terminates when all the input
117:  c          is exhausted.
118:  c
119: 425      $parend
120:  c          stop timing.
121:          $doall 430
122:          call timeout(itime)
123: 430      $parend
124:  c          compute timing in second.
125:          it=irep/10+1
126:          t(it)=itime(3)+.001*itime(4)
127:          if(it.eq.10)write(*,20)t
128: 410      continue
129: 400      continue
130:          $stop

```

```
131:    10    format(125a1)
132:    20    format(10f7.2)
133:        $end
```

```

1:      c
2:      c
3:      c
4:      $CHAREQU
5:      c
6:      c
7:      c
8:      c
9:      c
10:     c
11:     c
12:     c
13:     c
14:     c
15:     c
16:     character*1 string(500000),lpat(15),line(125)
17:     dimension ind(100),itime(144),t(10)
18:     dimension ldel0(0:127),ldel2(15),f(15)
19:     integer strlen,strmax,patlen,f
20:     integer large,ptl
21:     $shared string,itime,ind, strlen, patlen
22:     $shared nproc,nelem,ikey,np
23:     $region cr
24:     $usepar
25:     strmax=500000
26:     c
27:     c
28:     c
29:     c
30:     c
31:     c
32:     c
33:     c
34:     c
35:     c
36:     c
37:     c
38:     c
39:     c
40:     c
41:     c
42:     c
43:     c
44:     c
45:     c
46:     c
47:     c
48:     c
49:     c
50:     c
51:     c
52:     c
53:     c
54:     c
55:     c
56:     c
57:     c
58:     c
59:     c
60:     c
61:     c
62:     c
63:     c
64:     c
65:     c

```

*** Program 5.4 ***

the parallel version of the boyer-moore string searching algorithm.

two tables are precomputed. they are del0 and del2. the idea behind this method is that the search is started by comparing the leftmost character of pat and the patlen-th character of string instead of comparing the two rightmost characters of the two strings. for a more detailed discussion of the gains see boyer-moore's paper.

input the string of characters from mytext.

start random generator.

generate 100 random locations in string.

start timing

copy lpat from string starting from ikey.

```

66:          do 415 i=1,ptl
67:             ik=iky+i-1
68:          415      lpat(i)=string(ik)
69:          c          ldel0 setting-up
70:          do 100 i=0,127
71:          100      ldel0(i)=ptl
72:          do 101 i=1,ptl
73:          101      ldel0(ichar(lpat(i)))=ptl-i
74:          ldel0(ichar(lpat(ptl)))=large
75:          c          ldel2 setting up
76:          do 102 i=1,ptl
77:          102      ldel2(i)=2*ptl-i
78:          j=ptl
79:          k=ptl+1
80:          103      if(j.le.0) goto 106
81:             f(j)=k
82:          104      if(k.gt.ptl.or.lpat(j).eq.lpat(k))goto 105
83:             k=f(k)
84:             goto 104
85:          105      continue
86:             k=k-1
87:             j=j-1
88:             ldel2(k)=min0(ldel2(k),ptl-j)
89:             goto 103
90:          106      continue
91:          do 107 i=1,k
92:          107      ldel2(i)=min0(ldel2(i),ptl+k-i)
93:          c
94:          c          the boyer moore fast string matching algorithm.
95:          c
96:             i=ptl-1+is
97:          310      if(i.le.ie)goto 300
98:          c          input exhausted.
99:             goto 350
100:          c
101:          300      i=i+ldel0(ichar(string(i)))
102:          if(i.le.ie) goto 300
103:          c
104:          if(i.gt.large)goto 315
105:          c          input exhausted.
106:             goto 350
107:          315      i=i-large-1
108:             j=ptl-1
109:          c
110:          320      if(j.ne.0)goto 321
111:          c          an occurrence of pat is found
112:             i=i+2*ptl
113:             goto 310
114:          321      if(string(i).ne.lpat(j))goto 322
115:             j=j-1
116:             i=i-1
117:             goto 320
118:          322      continue
119:             mem=ldel0(ichar(string(i)))
120:             if(mem.eq.large) mem=0
121:             i=i+max0(mem,ldel2(j))
122:             goto 310
123:          350      continue
124:          c
125:          c          the algorithm terminates when the input
126:          c          is exhausted.
127:          c
128:          425      $parend
129:          c          stop timing
130:          $doall 435
131:             call timeout(itime)

```

```
132:    435    $parend
133:    c                compute timing in second.
134:                it=irep/10+1
135:                t(it)=itime(3)+.001*itime(4)
136:                if(it.eq.10)write(*,20)t
137:    410    continue
138:    400    continue
139:    $stop
140:    10    format(125a1)
141:    20    format(10f7.2)
142:    $end
```

```

1:      c
2:      c      *** Program 5.5 ***
3:      c
4:      $CHAREQU
5:      c      the parallel version of the Improved
6:      c      Boyer-Moore string searching algorithm.
7:      c
8:      c      two tables are precomputed. they are del0 and
9:      c      del2. the idea behind this method is that the
10:     c      search is started by comparing the leftmost
11:     c      character of pat and the patlen-th character of
12:     c      string instead of comparing the two rightmost
13:     c      characters of the two strings. for a more detailed
14:     c      discussion of the gains see boyer-moore's paper.
15:     c
16:     character*1 string(500000),lpat(15),line(125)
17:     dimension ind(100),itime(144),t(10)
18:     dimension ldel0(0:127),ldel2(15),f(15)
19:     integer  strlen,strmax,patlen,f
20:     integer  large,ptl
21:     $shared  string,itime,ind,strlen,patlen
22:     $shared  nproc,nelem,ikey,np
23:     $region  cr
24:     $usepar
25:     strmax=500000
26:     c      input the string of characters from mytext.
27:     strlen=1
28:     do 50 iline=1,4000
29:         read(*,10)line
30:         do 50 i=1,125
31:             string(strlen)=line(i)
32:             strlen=strlen+1
33:         if(strlen.gt.strmax) strlen=strmax
34:     c      find number of processors and name them.
35:     call nprocs (nproc)
36:     c      name the processors from 1 to nproc
37:     np=1
38:     $doall 60
39:     $enter cr
40:         ipath=np
41:         np=np+1
42:     $exit cr
43:     60  $parend
44:     c      start random generator.
45:     iran=ran(-1)
46:     do 400 patlen=6,15
47:     c      generate 100 random locations in string.
48:         do 405 i=1,100
49:             x=ran(1)
50:             405 ind(i)=1+(1.0-x)*(strlen+1-patlen)
51:         do 410 irep=1,100,10
52:             ikey=ind(irep)
53:     c      start timing
54:         $doall 420
55:         call timest
56:     420  $parend
57:     $doall 425
58:         nelem=strlen/nproc
59:         is=(ipath-1)*nelem+1
60:         ie=ipath*nelem+patlen-1
61:         if(ipath.eq.nproc) ie=strlen
62:         ptl=patlen
63:         iky=ikey
64:         large=strlen+100
65:     c      copy lpat from string starting from ikey.
66:         do 415 i=1,ptl

```

```

67:          ik=iky+i-1
68:      415      lpat(i)=string(ik)
69:      c          ldel0 setting-up
70:      do 100 i=0,127
71:      100      ldel0(i)=ptl
72:      do 101 i=1,ptl
73:      101          ldel0(ichar(lpat(i)))=ptl-i
74:          ldel0(ichar(lpat(ptl)))=large
75:      c          ldel2 setting up
76:      do 102 i=1,ptl
77:      102      ldel2(i)=2*ptl-i
78:      j=ptl
79:      k=ptl+1
80:      103      if(j.le.0) goto 106
81:          f(j)=k
82:      104      if(k.gt.ptl.or.lpat(j).eq.lpat(k))goto 105
83:          k=f(k)
84:          ldel2(k)=min0(ldel2(k),ptl-j)
85:          goto 104
86:      105      continue
87:          k=k-1
88:          j=j-1
89:          goto 103
90:      106      continue
91:      do 107 i=1,k
92:      107      ldel2(i)=min0(ldel2(i),ptl+k-i)
93:      c
94:      c      The improved Boyer-Moore fast string matching algorithm.
95:      c
96:          i=ptl-1+is
97:      310      if(i.le.ie)goto 300
98:      c          input exhausted.
99:          goto 350
100:      c
101:      300      i=i+ldel0(ichar(string(i)))
102:      if(i.le.ie) goto 300
103:      c
104:      if(i.gt.large)goto 315
105:      c          input exhausted.
106:          goto 350
107:      315      i=i-large-1
108:      j=ptl-1
109:      c
110:      320      if(j.ne.0)goto 321
111:      c          an occurrence of pat is found
112:      i=i+2*ptl
113:      goto 310
114:      321      if(string(i).ne.lpat(j))goto 322
115:          j=j-1
116:          i=i-1
117:          goto 320
118:      322      continue
119:      mem=ldel0(ichar(string(i)))
120:      if(mem.eq.large) mem=0
121:      i=i+max0(mem,ldel2(j))
122:      goto 310
123:      350      continue
124:      c
125:      c          the algorithm terminates when the input
126:      c          is exhausted.
127:      c
128:      425      $parend
129:      c          stop timing
130:      $doall 435
131:          call timeout(itime)
132:      435      $parend

```

```
133:      c      compute timing in second.
134:          it=irep/10+1
135:          t(it)=itime(3)+.001*itime(4)
136:          if(it.eq.10)write(*,20)t
137:      410      continue
138:      400      continue
139:      $stop
140:      10      format(125a1)
141:      20      format(10f7.2)
142:      $end
```

```

1:      c
2:      c      *** Program 6.1 ***
3:      c
4:      c      PARALLEL QUICKSORT ALGORITHM
5:      c      BREATH-FIRST METHOD.
6:      c
7:      c      The number of paths could be selected
8:      c      to P or greater than P.
9:      c
10:
11:      c*****
12:      c      S U B R O U T I N E   I N S E R T I O N   S O R T
13:      c*****
14:
15:      subroutine INSERT ( s,e)
16:      dimension a(16384)
17:      dimension lr(512,3),itime(144)
18:      integer s,e
19:      $shared a,itime,lr,n,m,nproc
20:      do 100 i=s+1,e
21:      v=a(i)
22:      j=i
23:      110      if( j.le.s.or.a(j-1).le.v) goto 120
24:                a(j)=a(j-1)
25:                j=j-1
26:                goto 110
27:      120      a(j)=v
28:      100      continue
29:      return
30:      end
31:
32:      c*****
33:      c      S O U B R O U T I N E   M E D I A N   O F   T H E   T H R E E
34:      c*****
35:
36:      subroutine MEDIAN (s,e)
37:      dimension a(16384),lr(512,3),itime(144)
38:      $shared a,itime,lr,n,m,nproc
39:      integer s,e
40:      c      THE PARTITIONING ELEMENT IS THE MEDIAN OF THE THREE.
41:      mid=(s+e)/2
42:      if( a(s).le.a(mid) ) goto 100
43:      sav=a(s)
44:      a(s)=a(mid)
45:      a(mid)=sav
46:      100      if( a(mid).le.a(e) ) goto 110
47:      sav=a(mid)
48:      a(mid)=a(e)
49:      a(e)=sav
50:      110      if( a(s).le.a(mid) ) goto 120
51:      sav=a(s)
52:      a(s)=a(mid)
53:      a(mid)=sav
54:      120      sav=a(s+1)
55:      a(s+1)=a(mid)
56:      a(mid)=sav
57:      return
58:      end
59:
60:      c*****
61:      c      S U B R O U T I N E   P A R T I T I O N
62:      c*****
63:
64:      subroutine PARTI (s,e,part)
65:      dimension a(16384),lr(512,3),itime(144)
66:      $shared a,itime,lr,n,m,nproc

```

```

67:      integer s,e,part
68:      call MEDIAN (s,e)
69:      i=s+1
70:      j=e
71:      v=a(i)
72:      100      if( j.le.i ) goto 110
73:      120      i=i+1
74:      if( a(i).lt.v ) goto 120
75:      130      j=j-1
76:      if( a(j).gt.v ) goto 130
77:      t=a(i)
78:      a(i)=a(j)
79:      a(j)=t
80:      goto 100
81:      110      t=a(i)
82:      a(i)=a(j)
83:      a(j)=a(s+1)
84:      a(s+1)=t
85:      part=j
86:      c      END OF THE PARTITIONING PROCESS.
87:      return
88:      end
89:
90:      c*****
91:      c      S U B R O U T I N E   Q U I C K S O R T
92:      c*****
93:
94:      subroutine QUICK (s,e)
95:      dimension a(16384)
96:      dimension lr(512,3),itime(144),stk(30)
97:      integer s,e,lr,stk,p
98:      $shared a,itime,lr,n,m,nproc
99:      p=3
100:      c      REPEAT UNTIL ( P = 1 ).
101:      100      continue
102:      if ( e+1-s .gt. m ) goto 110
103:      if ( e+1-s.gt.1 ) call INSERT (s,e)
104:      p=p-2
105:      s=stk(p)
106:      e=stk(p+1)
107:      goto 120
108:      110      call PARTI (s,e,j)
109:      c      THE LARGEST OF THE 2 SUB-SET IS PUSHED
110:      c      IN THE STACK FOR FURTHER PROCESSING.
111:      if( j-s.ge.e-j ) goto 140
112:      stk(p)=j+1
113:      stk(p+1)=e
114:      e=j-1
115:      p=p+2
116:      goto 120
117:      140      continue
118:      stk(p)=s
119:      stk(p+1)=j-1
120:      s=j+1
121:      p=p+2
122:      120      if( p.ne.1 ) goto 100
123:      return
124:      end
125:
126:      c*****
127:      c      S U B R O U T I N E   U P D A T E
128:      c*****
129:
130:      subroutine UPDAT (lev)
131:      dimension a(16384),lr(512,3),itime(144),slr(512,3)
132:      integer slr

```

```

133:      $shared a,itime,lr,n,m,nproc
134:      do 100 i=1,lev
135:      slr(i,1)=lr(i,1)
136:      slr(i,2)=lr(i,2)
137: 100    slr(i,3)=lr(i,3)
138:
139:      do 120 i=1,lev
140:      lr(2*i-1,1)=slr(i,1)
141:      lr(2*i-1,2)=slr(i,3)-1
142:      lr(2*i,1)=slr(i,3)+1
143: 120    lr(2*i,2)=slr(i,2)
144:      return
145:      end
146:
147: c*****
148: c                      M A I N   P R O G R A M
149: c*****
150:
151:      dimension a(16384),lr(512,3),itime(144)
152:      integer l,r,lr
153:      $shared a,itime,lr,n,m,nproc
154:      $usepar
155:      call nprocs (nproc)
156:
157:  c                      INITIATION OF SOME VARIABLES
158:      read(*,*)n,m,npaths
159:      l=1
160:      r=n
161:  c                      THE ARRAY A IS GENERATED RONDONLY.
162:      x=ran(-1)
163:      do 50 i=1,n
164: 50    a(i)=ran(1)
165:  c                      STRAT TIMING OF THE PARALLEL ALGORITHM.
166:      $doall 60
167:      call timest
168: 60    $parend
169:
170:      level=0
171:      lr(1,1)=1
172:      lr(1,2)=n
173:  c                      REPEAT UNTIL THE NUMBER OF NODES IS GREATER THAN NPROC
174: 100   continue
175:      lev2=2**level
176:      $dopar 110 ip=1,lev2
177:          l=lr(ip,1)
178:          r=lr(ip,2)
179:          call PARTI (l,r,j)
180:          lr(ip,3)=j
181: 110   $parend
182:  c                      UPDATE THE LR TABLE.
183:      call UPDAT (lev2)
184:      level=level+1
185:      lev2=2**level
186:
187:      if (lev2.lt.npaths) goto 100
188:      $dopar 130 ip=1,lev2
189:          l=lr(ip,1)
190:          r=lr(ip,2)
191:          call QUICK (l,r)
192: 130   $parend
193:  c                      STROP TIMING
194:      $doall 150
195:      call timout (itime)
196: 150   $parend
197:      call printt (itime)
198:      write(*,*)npaths

```

```
199:          do 500 i=1,n-1                                421
200:              if( a(i).gt.a(i+1)) write(*,*)' !!! ERROR !!!'
201:          500      continue
202:          $stop
203:          $end
```

```

1:      c
2:      c      *** Program 6.2 ***
3:      c
4:      c      PARALLEL QUICKSORT-MERGE ALGORITHM
5:      c
6:      c      The original set is divided into p subsets of n/P elements.
7:      c      Every subset is sorted using the SQ. Once all the p paths
8:      c      have completed the p subsets are the merged together.
9:      c      the number of subset could be P or more than p . It is
10:     c      selected through the var npaths.
11:
12:     c*****
13:     c      S U B R O U T I N E   M E R G E
14:     c*****
15:
16:     subroutine MERGE ( s1,e1,s2,e2)
17:     dimension a(16384),c(16384),itime(144)
18:     integer s1,e1,s2,e2
19:     $shared a,c,itime,n,m,nelem,np
20:
21:     i=s1
22:     j=s2
23:     k=s1
24: 100    if( i.gt.e1.or.j.gt.e2 ) goto 110
25:         if ( a(i).ge.a(j)) goto 120
26:             c(k)=a(i)
27:             i=i+1
28:             goto 130
29: 120    c(k)=a(j)
30:         j=j+1
31: 130    k=k+1
32:     goto 100
33: 110    continue
34:     if(i.le.e1) goto 140
35: 142    if( j.gt.e2) goto 144
36:         c(k)=a(j)
37:         k=k+1
38:         j=j+1
39:         goto 142
40: 144    continue
41:     goto 150
42: 140    if( i.gt.e1) goto 146
43:         c(k)=a(i)
44:         k=k+1
45:         i=i+1
46:         goto 140
47: 146    continue
48: 150    continue
49:     i=s1
50: 160    if ( i.ge.k ) goto 170
51:         a(i)=c(i)
52:         i=i+1
53:         goto 160
54: 170    continue
55:     return
56:     end
57:
58:     c*****
59:     c      S U B R O U T I N E   I N S E R T I O N   S O R T
60:     c*****
61:
62:     subroutine INSERT ( s,e)
63:     dimension a(16384),c(16384),itime(144)
64:     integer s,e
65:     $shared a,c,itime,n,m,nelem,np
66:     do 100 i=s+1,e

```

```

67:      v=a(i)
68:      j=i
69:      110      if( j.le.s.or.a(j-1).le.v) goto 120
70:                  a(j)=a(j-1)
71:                  j=j-1
72:                  goto 110
73:      120      a(j)=v
74:      100      continue
75:      return
76:      end
77:
78:  C*****
79:  C   S O U B R O U T I N E   M E D I A N   O F   T H E   T H R E E
80:  C*****
81:
82:      subroutine MEDIAN (s,e)
83:      dimension a(16384),itime(144),c(16384)
84:      $shared a,c,itime,n,m,nelem,np
85:      integer s,e
86:  C   THE PARTITIONNING ELEMENT IS THE MEDIAN OF THE THREE.
87:      mid=(s+e)/2
88:      if( a(s).le.a(mid) ) goto 100
89:          sav=a(s)
90:          a(s)=a(mid)
91:          a(mid)=sav
92:      100      if( a(mid).le.a(e) ) goto 110
93:          sav=a(mid)
94:          a(mid)=a(e)
95:          a(e)=sav
96:      110      if( a(s).le.a(mid) ) goto 120
97:          sav=a(s)
98:          a(s)=a(mid)
99:          a(mid)=sav
100:      120      sav=a(s+1)
101:          a(s+1)=a(mid)
102:          a(mid)=sav
103:          return
104:          end
105:
106:  C*****
107:  C   S U B R O U T I N E   P A R T I T I O N
108:  C*****
109:
110:      subroutine PARTI (s,e,part)
111:      dimension a(16384),itime(144),c(16384)
112:      $shared a,c,itime,n,m,nelem,np
113:      integer s,e,part
114:      call MEDIAN (s,e)
115:      i=s+1
116:      j=e
117:      v=a(i)
118:      100      if( j.le.i ) goto 110
119:      120      i=i+1
120:          if( a(i).lt.v ) goto 120
121:      130      j=j-1
122:          if( a(j).gt.v ) goto 130
123:          t=a(i)
124:          a(i)=a(j)
125:          a(j)=t
126:          goto 100
127:      110      t=a(i)
128:          a(i)=a(j)
129:          a(j)=a(s+1)
130:          a(s+1)=t
131:          part=j
132:  C

```

END OF THE PARTITIONNING PROCESS.

```

133:         return
134:         end
135:
136: c*****
137: c             S U B R O U T I N E   Q U I C K S O R T
138: c*****
139:
140:         subroutine QUICK (s,e)
141:         dimension a(16384),c(16384)
142:         dimension itime(144),stk(30)
143:         integer s,e,stk,p
144:         $shared a,c,itime,n,m,nelem,np
145:         p=3
146:
147: c             REPEAT UNTIL ( P = 1 ).
148: 100         continue
149:             if ( e+1-s .gt. m ) goto 110
150:                 if ( e+1-s.gt.1 ) call INSERT (s,e)
151:                     p=p-2
152:                     s=stk(p)
153:                     e=stk(p+1)
154:                     goto 120
155: 110         call PARTI (s,e,j)
156: c             THE LARGEST OF THE 2 SUB-SET IS PUSHED
157: c             IN THE STACK FOR FURTHER PROCESSING.
158:             if( j-s.ge.e-j ) goto 140
159:                 stk(p)=j+1
160:                 stk(p+1)=e
161:                 e=j-1
162:                 p=p+2
163:                 goto 120
164: 140         continue
165:                 stk(p)=s
166:                 stk(p+1)=j-1
167:                 s=j+1
168:                 p=p+2
169: 120         if( p.ne.1) goto 100
170:         return
171:         end
172:
173: c*****
174: c             M A I N   P R O G R A M
175: c*****
176:
177:         dimension a(16384),itime(144),c(16384)
178:         $shared a,c,itime,n,m,nelem,np
179:         $usepar
180: c             INITIATION OF SOME VARIABLES
181:         read(*,*)n,m,npaths
182:         nelem=n/npaths
183: c             THE ARRAY A IS GENERATED RONDONMLY.
184:         x=ran(-1)
185:         do 50 i=1,n
186: 50         a(i)=ran(1)
187: c             STRAT TIMING OF THE PARALLEL ALGORITHM.
188:         $doall 60
189:         call timest
190: 60         $parend
191:
192: c             DIVIDE THE ORIGINAL ARRAY INTO NPATHS SUB-ARRAYS AND
193: c             FORK NPATHS. EACH PATH PERFORMS THE SEQUENTIAL
194: c             QUICKSORT ALGORITHM.
195:         $dopar 100 ip=1,npaths
196:             is=(ip-1)*nelem+1
197:             ie=is+nelem-1

```

```

198:                if( ip.eq.npaths) ie=n
199:                call QUICK (is,ie)
200:    100          $parend
201:    c                PARALLEL MERGING
202:
203:                np=npaths/2
204:    110          if (np.lt.1) goto 120
205:                $dopar 130 ip=1,np
206:                    is1=2*(ip-1)*nelem+1
207:                    ie1=is1+nelem-1
208:                    is2=(2*ip-1)*nelem+1
209:                    ie2=is2+nelem-1
210:                    if (ip.eq.np) ie2=n
211:                    call MERGE (is1,ie1,is2,ie2)
212:    130          $parend
213:                np=np/2
214:                nelem=nelem*2
215:                goto 110
216:    120          continue
217:    c                STOP TIMING
218:                $doall 150
219:                call timeout (itime)
220:    150          $parend
221:                call printt (itime)
222:                write(*,*)npaths
223:                do 500 i=1,n-1
224:                    if (a(i).gt.a(i+1)) write(*,*)' !!! ERROR !!!'
225:    500          continue
226:                $stop
227:                $end

```

```

1:      C
2:      C      *** Program 6.3 ***
3:      C
4:      C      PARALLEL BOUNDED-PARTITIONED SORTING ALGORITHM
5:      C
6:      C
7:      C The original set is partitioned into P sub-sets in parallel
8:      C P+1 elements, a(1), a(nelem),...a(i*nelem)...a(n), are selected
9:      C and sorted. If processors are number from 1 to P and if
10:     C every processor ip picks all elmts that lay between U(ip-1)
11:     C and U(ip) then the P subsets are surely independent and
12:     C can be sorted in parallel. however there is only one drawback
13:     C which is can affect the overall performance is that at the
14:     C sorted P subsets are to copied back in the original set only
15:     C when all the all P processors have completed sorting their
16:     C sub-sets.
17:     C      Number of Paths must be greater than 1.
18:
19: C*****
20: C      S U B R O U T I N E      I N S E R T I O N      S O R T
21: C*****
22:
23:     subroutine INSERT ( s,e)
24:     dimension a(16384),itime(144),c(16384),dem(65),len(10,2)
25:     dimension b(16384)
26:     $shared a,b,itime,n,m,nproc,nelem,np,npa,dem,len
27:     common /CCCC/ c
28:     integer s,e
29:     do 100 i=s+1,e
30:     v=c(i)
31:     j=i
32: 110         if( j.le.s.or.c(j-1).le.v) goto 120
33:             c(j)=c(j-1)
34:             j=j-1
35:             goto 110
36: 120         c(j)=v
37: 100     continue
38:     return
39:     end
40:
41: C*****
42: C      S O U B R O U T I N E      M E D I A N      O F      T H E      T H R E E
43: C*****
44:
45:     subroutine MEDIAN (s,e)
46:     dimension a(16384),itime(144),c(16384),dem(65),len(10,2)
47:     dimension b(16384)
48:     $shared a,b,itime,n,m,nproc,nelem,np,npa,dem,len
49:     common /CCCC/ c
50:     integer s,e
51:     C      THE PARTITIONING ELEMENT IS THE MEDIAN OF THE THREE.
52:     mid=(s+e)/2
53:     if( c(s).le.c(mid) ) goto 100
54:     sav=c(s)
55:     c(s)=c(mid)
56:     c(mid)=sav
57: 100     if( c(mid).le.c(e) ) goto 110
58:     sav=c(mid)
59:     c(mid)=c(e)
60:     c(e)=sav
61: 110     if( c(s).le.c(mid) ) goto 120
62:     sav=c(s)
63:     c(s)=c(mid)
64:     c(mid)=sav
65: 120     sav=c(s+1)
66:     c(s+1)=c(mid)

```

```

67:      c(mid)=sav
68:      return
69:      end
70:
71:  c*****
72:  c      S U B R O U T I N E   P A R T I T I O N
73:  c*****
74:
75:      subroutine PARTI (s,e,part)
76:      dimension a(16384),itime(144),c(16384),dem(65),len(10,2)
77:      dimension b(16384)
78:      $shared a,b,itime,n,m,nproc,nelem,np,npa,dem,len
79:      common /CCCC/ c
80:      integer s,e,part
81:      call MEDIAN (s,e)
82:      i=s+1
83:      j=e
84:      v=c(i)
85:  100      if( j.le.i ) goto 110
86:  120          i=i+1
87:          if( c(i).lt.v ) goto 120
88:  130          j=j-1
89:          if( c(j).gt.v ) goto 130
90:          t=c(i)
91:          c(i)=c(j)
92:          c(j)=t
93:          goto 100
94:  110          t=c(i)
95:          c(i)=c(j)
96:          c(j)=c(s+1)
97:          c(s+1)=t
98:          part=j
99:  c
100:      return
101:      end
102:
103:  c*****
104:  c      S U B R O U T I N E   Q U I C K S O R T
105:  c*****
106:
107:      subroutine QUICK (s,e)
108:      dimension a(16384),itime(144),c(16384),dem(65),len(10,2)
109:      dimension b(16384)
110:      $shared a,b,itime,n,m,nproc,nelem,np,npa,dem,len
111:      common /CCCC/ c
112:      dimension stk(30)
113:      integer s,e,stk,p
114:      p=3
115:  c      REPEAT UNTIL ( P = 1 ).
116:  100      continue
117:      if ( e+1-s .gt. m ) goto 110
118:          if ( e+1-s.gt.1 ) call INSERT (s,e)
119:          p=p-2
120:          s=stk(p)
121:          e=stk(p+1)
122:          goto 120
123:  110      call PARTI (s,e,j)
124:  c      THE LARGEST OF THE 2 SUB-SET IS PUSHED
125:  c      IN THE STACK FOR FURTHER PROCESSING.
126:      if( j-s.ge.e-j ) goto 140
127:          stk(p)=j+1
128:          stk(p+1)=e
129:          e=j-1
130:          p=p+2
131:          goto 120
132:  140      continue

```

```

133:          stk(p)=s
134:          stk(p+1)=j-1
135:          s=j+1
136:          p=p+2
137: 120      if( p.ne.1) goto 100
138:          return
139:          end
140:
141: c*****
142: c          M A I N   P R O G R A M
143: c*****
144:
145:          dimension a(16384),itime(144),c(16384),dem(65),len(10,2)
146:          dimension b(16384)
147:          $shared a,b,itime,n,m,nproc,nelem,np,npa,dem,len
148:          $region cr
149:          common /CCCC/ c
150:          integer slen
151:          $usepar
152:
153:  c          INITIATION OF SOME VARIABLES
154:          read(*,*)n,m,npaths
155:  c          Name the processors from 1...nproc
156:          nproc=0
157:          $doall 100
158:              $enter cr
159:                  nproc=nproc+1
160:                  ip=nproc
161:              $exit cr
162: 100      $parend
163:          nelem=n/npaths
164:  c          THE ARRAY A IS GENERATED RONDONLY.
165:          x=ran(-1)
166:          do 110 i=1,n
167: 110      a(i)=ran(1)
168:  c          START TIMING OF THE PARALLEL ALGORITHM.
169:          $doall 120
170:              call timest
171: 120      $parend
172:
173:
174:  c          Define p+1 elements. (1+ip*n/npaths)
175:          dem(1)=a(1)
176:          do 130 ii=2,npaths
177:              ik=(ii-1)*nelem
178:              dem(ii)=a(ik)
179: 130      continue
180:          dem(npaths+1)=a(n)
181:  c          Sort dem array.
182:          do 140 ii=1,npaths+1
183:              v=dem(ii)
184:              ij=ii
185: 150          if( ij.le.1.or.dem(ij-1).le.v) goto 160
186:                  dem(ij)=dem(ij-1)
187:                  ij=ij-1
188:                  goto 150
189:          dem(ij)=v
190: 160      continue
191:
192:          slen=1
193:          np=0
194:          npa=npaths
195: 170      if( npaths.le.0) goto 180
196:          $doall 190
197:              U1=dem(np+ip)
198:              U2=dem(np+ip+1)

```

```

199:      c      Pick all the elements that lay between U1 and U2
200:      ik=0
201:      if(np+ip.ne.1) goto 200
202:      c      Case when ip = 1.
203:      do 210 ii=1,n
204:      v=a(ii)
205:      if (.not.(v.le.U2)) goto 220
206:      ik=ik+1
207:      c(ik)=v
208:      220      continue
209:      210      continue
210:      goto 230
211:      200      continue
212:      if( np+ip.ne.npa) goto 240
213:      c      Case when ip = NPATH
214:      do 250 ii=1,n
215:      v=a(ii)
216:      if (.not.(U1.lt.v)) goto 260
217:      ik=ik+1
218:      c(ik)=v
219:      260      continue
220:      250      continue
221:      goto 230
222:      240      continue
223:      c      Case when ip # 1 and nproc.
224:      do 270 ii=1,n
225:      v=a(ii)
226:      if(.not.(U1.lt.v.and.v.le.U2)) goto 280
227:      ik=ik+1
228:      c(ik)=v
229:      280      continue
230:      270      continue
231:      230      continue
232:      is=1
233:      ie=ik
234:      len(ip,1)=ik
235:      c      save the number of elements picked in len
236:      call QUICK (is,ie)
237:      190      $parend
238:
239:      c      copy back all the sorted elements (is,ie)
240:      len(1,2)=slen
241:      do 290 ii=2,nproc
242:      290      len(ii,2)=len(ii-1,1)+len(ii-1,2)
243:      slen=len(nproc,1)+len(nproc,2)
244:
245:      $doall 300
246:      ik=len(ip,2)-1
247:      ie=len(ip,1)
248:      do 310 ii=1,ie
249:      310      b(ik+ii)=c(ii)
250:      300      $parend
251:
252:      npaths=npaths-nproc
253:      np=np+nproc
254:      goto 170
255:      180      continue
256:      c      copy in parallel b into a
257:      nelem=n/nproc
258:      $doall 320
259:      is=(ip-1)*nelem+1
260:      ie=is+nelem-1
261:      if(ip.eq.nproc) ie=n
262:      do 330 ii=is,ie
263:      a(ii)=b(ii)
264:      330      continue

```

```
265: 320 $parend
266: c      stop timing for the sorting phase.
267:      $doall 340
268:      call timout (itime)
269: 340 $parend
270:      call printt(itime)
271:      write(*,*)npaths
272: c      check for correctness
273:      do 500 i=1,n-1
274:          if(a(i).gt.a(i+1)) write(*,*)i,i+1,a(i),a(i+1)
275: 500 continue
276: $stop
277: $end
```

```

1:
2:      c
3:      c
4:      c
5:      c
6:      c
7:      c If the range [a,b] is known then the set is partitioned into P
8:      c sub-sets producing better distribution than the PBPS.pf
9:      c If processors are number from 1 to P and if
10:     c every processor ip picks all elements that lay between
11:     c  $a+(ip-1)*(b-a)/P$  and  $a+ip*(b-a)/P$  then p independent subsets
12:     c are obtained. These p subsets can be sorted in parallel by p.
13:     c However there is only one drawback which is going to affect
14:     c the overall performance .
15:     c the p subsets are to copied back to the original array.
16:
17: c*****
18: c      S U B R O U T I N E      I N S E R T I O N      S O R T
19: c*****
20:
21:     subroutine INSERT ( s,e)
22:     dimension a(16384),itime(144),c(16384),dem(65),len(10,2)
23:     dimension b(16384)
24:     $shared a,b,itime,n,m,nproc,nelem,np,npa,dem,len
25:     common /CCCC/ c
26:     integer s,e
27:     do 100 i=s+1,e
28:     v=c(i)
29:     j=i
30: 110         if( j.le.s.or.c(j-1).le.v) goto 120
31:             c(j)=c(j-1)
32:             j=j-1
33:             goto 110
34: 120         c(j)=v
35: 100     continue
36:     return
37:     end
38:
39: c*****
40: c      S O U B R O U T I N E      M E D I A N      O F      T H E      T H R E E
41: c*****
42:
43:     subroutine MEDIAN (s,e)
44:     dimension a(16384),itime(144),c(16384),dem(65),len(10,2)
45:     dimension b(16384)
46:     $shared a,b,itime,n,m,nproc,nelem,np,npa,dem,len
47:     common /CCCC/ c
48:     integer s,e
49:     c      THE PARTITIONING ELEMENT IS THE MEDIAN OF THE THREE.
50:     mid=(s+e)/2
51:     if( c(s).le.c(mid) ) goto 100
52:     sav=c(s)
53:     c(s)=c(mid)
54:     c(mid)=sav
55: 100     if( c(mid).le.c(e) ) goto 110
56:     sav=c(mid)
57:     c(mid)=c(e)
58:     c(e)=sav
59: 110     if( c(s).le.c(mid) ) goto 120
60:     sav=c(s)
61:     c(s)=c(mid)
62:     c(mid)=sav
63: 120     sav=c(s+1)
64:     c(s+1)=c(mid)
65:     c(mid)=sav
66:     return

```

```

67:      end
68:
69:      c*****
70:      c          S U B R O U T I N E   P A R T I T I O N
71:      c*****
72:
73:      subroutine PARTI (s,e,part)
74:      dimension a(16384),itime(144),c(16384),dem(65),len(10,2)
75:      dimension b(16384)
76:      $shared a,b,itime,n,m,nproc,nelem,np,npa,dem,len
77:      common /CCCC/ c
78:      integer s,e,part
79:      call MEDIAN (s,e)
80:      i=s+1
81:      j=e
82:      v=c(i)
83:      100      if( j.le.i ) goto 110
84:      120          i=i+1
85:                  if( c(i).lt.v ) goto 120
86:      130          j=j-1
87:                  if( c(j).gt.v ) goto 130
88:                  t=c(i)
89:                  c(i)=c(j)
90:                  c(j)=t
91:                  goto 100
92:      110          t=c(i)
93:                  c(i)=c(j)
94:                  c(j)=c(s+1)
95:                  c(s+1)=t
96:                  part=j
97:      c
98:      return
99:      end
100:
101:      c*****
102:      c          S U B R O U T I N E   Q U I C K S O R T
103:      c*****
104:
105:      subroutine QUICK (s,e)
106:      dimension a(16384),itime(144),c(16384),dem(65),len(10,2)
107:      dimension b(16384)
108:      $shared a,b,itime,n,m,nproc,nelem,np,npa,dem,len
109:      common /CCCC/ c
110:      dimension stk(30)
111:      integer s,e,stk,p
112:      p=3
113:      c          REPEAT UNTIL ( P = 1 ).
114:      100      continue
115:      if ( e+1-s .gt. m ) goto 110
116:      if ( e+1-s.gt.1 ) call INSERT (s,e)
117:      p=p-2
118:      s=stk(p)
119:      e=stk(p+1)
120:      goto 120
121:      110      call PARTI (s,e,j)
122:      c          THE LARGEST OF THE 2 SUB-SET IS PUSHED
123:      c          IN THE STACK FOR FURTHER PROCESSING.
124:      if( j-s.ge.e-j ) goto 140
125:      stk(p)=j+1
126:      stk(p+1)=e
127:      e=j-1
128:      p=p+2
129:      goto 120
130:      140      continue
131:      stk(p)=s
132:      stk(p+1)=j-1

```

```

133:          s=j+1
134:          p=p+2
135: 120      if( p.ne.1) goto 100
136:          return
137:          end
138:
139: c*****
140: c          M A I N   P R O G R A M
141: c*****
142:
143:          dimension a(16384),itime(144),c(16384),dem(65),len(10,2)
144:          dimension b(16384)
145:          $shared a,b,itime,n,m,nproc,nelem,np,npa,dem,len
146:          $region cr
147:          common /CCCC/ c
148:          integer slen
149:          $usepar
150:
151:  c          INITIATION OF SOME VARIABLES
152:          read(*,*)n,m,npaths
153:  c          Name the processors from 1...nproc
154:          nproc=0
155:          $doall 100
156:              $enter cr
157:                  nproc=nproc+1
158:                  ip=nproc
159:              $exit cr
160: 100      $parend
161:          nelem=n/npaths
162:  c          THE ARRAY A IS GENERATED RONDONLY.
163:          x=ran(-1)
164:          do 110 i=1,n
165: 110      a(i)=ran(1)
166:  c          START TIMING OF THE PARALLEL ALGORITHM.
167:          $doall 120
168:              call timest
169: 120      $parend
170:
171:  c          The range is supposed to be known and it is [0,1]
172:          dem(1)=0.00
173:          do 130 ii=2,npaths
174:              dem(ii)=(ii-1)*1.00/npaths
175: 130      continue
176:          dem(npaths+1)=1.00
177:
178:          slen=1
179:          np=0
180:          npa=npaths
181: 170      if( npaths.le.0) goto 180
182:          $doall 190
183:              U1=dem(np+ip)
184:              U2=dem(np+ip+1)
185:  c          Pick all the elements that lay between U1 and U2.
186:              ik=0
187:                  do 270 ii=1,n
188:                      v=a(ii)
189:                      if(.not.(U1.lt.v.and.v.le.U2)) goto 280
190:                          ik=ik+1
191:                          c(ik)=v
192: 280                  continue
193: 270                  continue
194:                      is=1
195:                      ie=ik
196:                      len(ip,1)=ik
197:  c          save the number of elements picked in len
198:          call QUICK (is,ie)

```

```

199:      190      $parend
200:
201:      c          copy back all the sorted elements (is,ie)
202:      len(1,2)=slen
203:      do 290 ii=2,nproc
204:      290          len(ii,2)=len(ii-1,1)+len(ii-1,2)
205:      slen=len(nproc,1)+len(nproc,2)
206:
207:      $doall 300
208:          ik=len(ip,2)-1
209:          ie=len(ip,1)
210:          do 310 ii=1,ie
211:      310              b(ik+ii)=c(ii)
212:      300      $parend
213:
214:      npaths=npaths-nproc
215:      np=np+nproc
216:      goto 170
217:      180      continue
218:      c          copy in parallel b into a
219:      nelem=n/nproc
220:      $doall 320
221:          is=(ip-1)*nelem+1
222:          ie=is+nelem-1
223:          if(ip.eq.nproc) ie=n
224:          do 330 ii=is,ie
225:              a(ii)=b(ii)
226:      330      continue
227:      320      $parend
228:      c          stop timing for the sorting phase.
229:      $doall 340
230:          call timeout (itime)
231:      340      $parend
232:      call printt(itime)
233:      write(*,*)npaths
234:      c          check for correctness
235:      do 500 i=1,n-1
236:          if(a(i).gt.a(i+1)) write(*,*)' !!! ERROR2 !!!'
237:      500      continue
238:      $stop
239:      $end

```

Appendix B

SUMMARY OF THE OCCAM LANGUAGE

In OCCAM processes are connected to form concurrent systems, each process can be regarded as a black box with an internal state which can communicate with other processes via point to point communication channels. The processes themselves are finite. Each process starts, performs a number of actions then terminates. An action may be a set of parallel processes to be performed at the same time. As a process is itself composed of processes which may themselves be executed in parallel, a process allows internal concurrency which varies with time.

Processes

All processes are constructed from three primitive processes, assignment, input and output.

Assignment: An assignment is indicated by the symbol `:=`, for example, `v:=e` sets variable `v` to the value of the expression `e` and then terminates.

Input: An input is indicated by the symbol `?`, for example, `c?x` inputs a value from a channel `c` assigning it to `x` and then terminating.

Output: An output is indicated by the symbol `!` and `c!e` outputs the expression `e` to channel `c`, and then terminates.

A pair of concurrent processes communicate using a one way channel connecting the two processes. One process outputs a message to the channel, the other process inputs the message from the channel. A particular process can be ready to communicate on one or more of its channels any time between its start and termination, but a communication only takes place when both it and the process sharing

one of its channels is ready. Where a number of connected processes are ready simultaneously communication can occur in parallel.

Constructs:

A number of processes can be combined to form a construct which is itself a process and can be used as a component for other constructs. Each component process is indented by two spaces from the left hand margin indicating which construct it is part of. There are only four basic construct types: sequential, parallel, conditional and alternative.

SEQ: is the keyword for a sequential construct denoted

```

SEQ
  P1
  P2
  P3
  ...

```

where the component processes P_1 , P_2 , P_3 , ... are executed in strict sequence with process P_i finishing before P_{i+1} starts and after P_{i-1} terminates. Sequential constructs are similar to programs written in conventional programming languages.

PAR: is the keyword for a parallel construct of the form

```

PAR
  P1
  P2
  P3
  ...

```

and in contrast to SEQ, here all the component processes P_1 , P_2 , P_3 , ... are executed concurrently. The PAR construct terminates when all the component processes have finished.

IF: is the keyword for a conditional construct with the appearance

```

IF
    condition 1
        P1
    condition 2
        P2
    ...

```

This means that P_1 is executed if condition 1 is true, otherwise P_2 if condition 2 is true, etc. Notice the strict sequential ordering of tests. Only one of the processes P_i is executed and the IF construct terminates when the process finishes.

ALT: is the keyword for the alternative construct

```

ALT
    input 1
        P1
    input 2
        P2
    ...

```

This construct waits until one of input 1, input 2, input 3, ... is ready. If input 1 is ready first, input 1 is performed and on completion P_1 is executed. Similarly if input i is ready first input i is performed and P_i executed. Only one of the inputs is

performed and its corresponding process executed before the construct terminates. If more than one input becomes ready at the same time the one executed is chosen arbitrarily.

Repetition:

There is only one explicit construction for repetition denoted by

$$\begin{array}{c} \text{WHILE condition} \\ P \end{array}$$

which repeatedly executes process P until the value of the condition is false. Observe that P itself can be a composition of sequential and parallel constructs.

Replication:

A replicator is used with a constructor to replicate the component process a number of times. With SEQ a standard for loop

$$\begin{array}{c} \text{SEQ } i=[0 \text{ FOR } n] \\ P \end{array}$$

is created executing process P sequentially n times. When used with PAR an array of concurrent processes with the form

$$\begin{array}{c} \text{PAR } i=[0 \text{ FOR } n] \\ P_i \end{array}$$

is created such that n similar processes P_0, P_1, \dots, P_{n-1} are executed in parallel. Notice that $i=0(1)n-1$ not n , thus if generally $i[\text{base FOR count}]$ there are $\text{base}+\text{count}-1$ values i takes starting with $i=\text{base}$.

Declarations:

A declaration introduces a new identifier for use in the process that follows it, and defines the meaning the identifier will have within the process. If the new identifier is the same as one already in use, all subsequent occurrences of the identifier in the process will refer to the meaning of the most recent declaration. Declarations are of four basic types VAR, CHAN, DEF and PROC linked to a following process by a colon (:) at the last line of the declaration. The process follows on the next line at the same level of indentation as the keyword declaration. For example:

```
VAR x:
```

```
  P
```

declares variable x to be used in process P, and

```
CHAN C:
```

```
  P
```

defines a channel C to be used in communication for P. A variable vector declaration introduces an identifier to be used as a vector of variables, viz:

```
VAR list [16]:
```

```
  P
```

for a vector named list of 16 variables indexed as list[0], list[1], ... list [15]. Likewise a channel vector declaration introduces a new identifier as a vector of channels for communicating between concurrent processes

```
CHAN C[n]:
```

```
P
```

DEF associates a name with a constant value, or with a table of constant values, e.g.

```
DEF a=1, b=2:
```

associating a with 1 and b with 2, using these identifiers within a process yields the associated values.

The PROC declaration introduces an identifier to name the process which follows, indented, on the succeeding lines. The process is termed the named process and is itself followed by a process in which the named process will be used. The named process can have parameters which are declared with the declaration of the named process and are called formal parameters. The named process text will be substituted for all occurrences of the process name in subsequent processes, the var and chan variables substituted in place of the formal parameters are called actual parameters. For example,

```
PROC buffer(CHAN in,out) =
    WHILE TRUE
        VAR x :
        SEQ
            in?x
            out!x :
        CHAN c,c1,c2 :
        PAR
            buffer(c1,c)
            buffer(c,c2)
```

declares two buffer processes executed concurrently, `buffer` is the named process with formal channel parameters `in` and `out`. In the following process `C,C1,C2` are actual parameters and on execution the `WHILE` loop will be textual substituted for occurrence of the name `buffer` and `C,C1,C2` substituted for `in` and `out` respectively. The size of a vector is not specified in the formal parameters of a named process and different sized vectors may be used as actual parameters on different substitutions. In addition to the standard declarations `VAR` and `CHAN`, a `VALUE` parameter may also be used, as either an ordinary or vector formal parameter and cannot be changed within a process by assignment or input.

Finally an identifier which is used but not declared in a named process is termed a free identifier. Any free identifier in use when a named process substitution takes place must be the same as a variable already in use. The free variable then takes on the most recent incarnation of the variable at the point where the process substitution takes place.

Program Format:

In OCCAM indentation from the left hand margin indicates program structure. Each process starts on a new line, at an indentation level indicated by the following rules.

Constructs:

The construct keyword (and the optional replicator) occupies the first line. Each of the component processes start on a new line and are indented by two spaces more than the keyword.

Conditionals:

The condition expression occupies the first line, and the component process starts on the next line indented by two more spaces.

Alt inputs:

The expression and its associated input occupy the first line and the component process starts on the next line indenting two more spaces.

Declarations:

Each declaration starts on a new line, at the same level of indentation as the process it prefixed, the final line of the declaration being terminated by a colon. Blank lines can be inserted anywhere and are ignored.

A construct can be broken to occupy more than one line, with line breaks occurring after comma, semicolon and before the second operand of an operator (requiring two operands). The continued line must be more indented than the first line of the construct.

Comments:

Comments are denoted by double hyphen (--) and terminate at the end of a line. All characters of a comment are ignored. A comment may follow an OCCAM construct on the same line or be on a line by itself.

This summary of OCCAM is taken from INMOS [84,85] and implements 'proto-OCCAM'. A more sophisticated version OCCAM 2 is now available providing Real, Integer, and Boolean types. We remark that the programming in this thesis was performed on the Sequent Balance 2000 machine under UNIX using Loughborough OCCAM as

implemented by R P Stallard. Appendix C discusses the Loughborough version of OCCAM and particularly its extensions of proto-OCCAM to allow real variables and non-standard OCCAM features. We point out that the systolic programs listed in Appendix D where possible have avoided these non-standard characteristics.

Appendix C

LOUGHBOROUGH OCCAM COMPILER VERSION 5.0 DOCUMENTATION

Help for running the occam compiler

A source 'occam' file (OCCAM and INMOS are trademarks of the INMOS group of companies) must be of the form '*.occ', to compile it to form an 'a.out' command file use the default options. For example to compile 'my_first.occ' :-

```
occam my_first.occ
```

An executable object 'a.out' is produced. As a shortcut you can omit the '.occ' affix and just say 'occam my_first', the compiler will add on the affix for you.

If a program is split into several files these can be separately compiled and linked together using the occam compiler and built in linker.

Each previously compiled occam program is specified in the command line in the form '*.o' e.g. :-

```
occam main.occ numericlib.o screenlib.o
```

This will compile the source of 'main' and link it in with the pre compiled library occam files 'numericlib.occ' 'screenlib.occ'. The -l option is used to generate new versions of library file objects.

Various switch options are provided, mainly for compiler debugging. Flags can either be put separately ('-g -l') or together and in any order ('-lg', '-gl'). The following switches may be useful :-

```
-g :  
    occam -g fast.occ
```

Compile the occam program as before but run the resulting program immediately (a compile, load and go option). If flag options are specified that apply to the run of the program these will be passed on as in 'occam -gqc fast'.

```
-l :  
    occam -l new_lib
```

Compile the program and produce object but do not link the object files together to produce an object program. This option is used for building up libraries of routines or to cut down the compilation time for compiling one long program.

```
-o :  
    occam keep_it -o saverun
```

Compile the program as normal but place the object program in the file 'saverun' rather than the default 'a.out'. Useful for saving several occam object files at the same time.

```
-x :  
    occam -x old_fashioned.occ
```

Compile according to the strict Inmos occam specification, LUT extensions (see file 'occamversion') currently include :-

- Multiple source file cross linking.
- Dynamic features.
- Variable PAR replicator counts.
- Floating point arithmetic.

```
-c :  
    a.out -c
```

Run the object program with cursor addressable facilities enabled, the standard library procedures 'goto.x.y' and 'clear.screen' require these facilities.

```
-G :    occam -G error_prone
```

Compiles the file as normal but generates a symbol file as well (in this case it would be 'error_prone.sym'), this is used by the run-time system to inspect the values of variables.

```
-q :  
    a.out -q
```

Run the object program without producing any characters to the screen other than those output by the program (unless CTRL c used). This enables occam programs to dump output that can be processed by other occam programs.

```
-F and -M :  
    occam -F num.occ
```

'-P' Includes the floating point library routines to provide a simple real number arithmetic capability. '-M' includes both the floating point and mathematical library routines to provide mathematical library routines.

```
-I :
```

This provides the features of the Inmos proto-occam definition (see 'occam version') such as STOP and TIME, it should be used where possible as it is closer to the occam-2 definition.

Full list of compiler option flags

The full (often cryptic) range of switch options are as follows. Several switch flags can be given, in any order and either separately or together. The mnemonic character giving the switch is highlighted by a capital letter. They are divided into sections - user defined flags, and system defined options, which are selected by prefixing with '%'.
User Flags

- % The next flag(s) are system flags - switch flag mode.
- c Run the program with Cursor addressable options enabled. The library routines 'clear.screen' and 'goto.x.y' need this flag set. If used for the compiler must also give the -g option.
- e Produce object/run object for Execution tracing. The resulting object file is then run with the '-e' option. This utility is described in 'tracerinfo'.
- f Force full occam semantic check on use of variables. A variable (not vectors though) can not be set within a PAR construct if the declaration is outside the PAR. This applies equally to procedure calls that change global variables.
- g Run the resulting object file if compilation succeeded. The program Goes immediately it is ready to.
- h Print out this 'Help' information.
- i Force an Interrupt immediately before start of execution - immediately displays the debug help menu. This enables break and trace points to be setup prior to anything being executed.
- l Compile but do not link the occam source. Needed when using multiple occam source Library files.
- m Check that every channel Match properly on execution, channels can have only one input and one output process during execution.
- o Produce an Object program with name given by the non-switch argument following this switch. Enables you to choose an object file name other than 'a.out'.
- q Run the program without outputting some non occam program produced messages - e.g. 'OCCAM Start Run'. Must give -g option as well 'q' stands for Quiet. Useful when producing output to be piped or processed by other programs.

- w Suppress the Warning messages from the compiler - when you have seen these warnings once you may find it less irritating to suppress them on subsequent compilations - does not affect error reporting or any other compiler action.
- x Do not permit any local LUT extensions in the source text. See 'occinfo' for information about these - for example recursion and EXTERNAL procedure definitions. Useful if moving an occam program for use on another occam compiler system.
- F Include the standard Floating point library routines. Provides routines to read or write floating point routines to channels.
- G Produce a symbol table file (with affix '.sym') for use with the 'm' option in the dynamic debugger for symbol value examination.
- I Permit the use of INMOS proto-occam version 2. These changes include the use of 'TIME' instead of 'NOW', the 'STOP' primitive and the use of 'Stopping IF' - an alternative without any TRUE conditions will STOP.
- L Use Long winded load, all the 'C' libraries are added at the last moment rather than using the pre-linked object, this may be useful if a user occam/C library calls a 'C' routine that is not used in the occam run time system. See 'libraryhelp' for more info.
- M Include the Mathematical library and floating point routines.
- O Produce optimized object. May improve run time by 20%.
- R Use Randomized scheduling when running the program - the same scheduler choices will not be made on separate executions. This gives non-deterministic execution and will be slightly slower but may be useful occasionally.
- S Do not include the Standard I/O routines with the object. This library is included by default, there is no reason not to want to include it unless you want to devise a totally new one.
- T The next argument is a Timing definition file built by the 'timebuild' utility to be used in conjunction with the '-e' option, supplying '-T' automatically selects '-e'. If this option is not selected the execution timings are taken from the source library file 'times'. Look at the 'timerinfo' help file for more details.
- V The compiler will normally desist reporting errors and warnings after the first fifty or so, with this option all the errors will be reported. May produce Very Verbose output.
- W Give Warning messages about declarations that turn out not to have been used at all. This may highlight misspelt declarations or existence of no longer used procedures.

System Flags

- % Switch back to expecting 'user' mode flag options.
This means you can replace -G%v by -%v%G.
- a Enable Analysis of the usage of channels - this facility is still under test.
- n Check the source occam for syntax errors, but do Not produce any object data from it.
- t Print out the program in the form just after it was Transformed.
Not generally useful as the program has changed so much.
- v Give Verbose information at each stage when running the compiler - will print out a more accurate description of the system commands it is calling and all the files it accesses.
Also switches on a full print out of the occam link information.
- A Produce the object code ('C' or Assembler) in a permanent file so that it can be inspected.
- C Produce 'C' rather than assembler output from the occam compiler then compile and link it. There will be *.o and *.c containing the object and compiler generated source created in the directory.
The 'C' and assembler code produced will be similar and there is little point in producing 'C' unless to waste time ! (as the 'C' compilation phase takes a long time). If the compiler is ported to a non-VAX system then this option will automatically be selected.
- D Switch on variable name and line number Dumping in the C/Assembler 'object' file so that the object can be tied in with the source.
- H Undocumented feature under test.
- L Produce an occam-'C' interface Library, the two files ending '-c.c' and '.occ' are linked together, the occam can refer directly to the 'C' routines.
- N Run the compiler showing the steps it would execute but without actually doing anything - like '-n' in the UNIX 'make' command. Useful when options start getting complicated. A No operation facility.
- Q Undocumented feature under test.
- S Do not apply some Simplifying transformations on the program. These currently remove constructs with no processes in them and redundant SEQ and PAR headers. These save a small amount of space and time at run and compile time and there is little point in turning off this option.
- X Print out the procedures that have been defined in the link files but has not been referenced - detects extra procedures defined across files but not used.
- Y Produce the linker assembler output in a permanent file rather than in a temporary file on '/tmp'. Enables the output from the linker to be debugged.
- Z Get the linker to print out all the definitions it is told about.

Description of the library routines

Standard Library

Provide commonly used routines to read and write to the keyboard and screen channels. The routines are written in 'C' and occam and use standard C or 'curses' I/O routines. There are also general routines for use to pause or abort a program as well as to use the 'C' random number routines. They are available by default to all programs unless the -S compiler flag is used to override their inclusion.

EXTERNAL PROC str.to.screen (VALUE s []) :

Output the string s (a byte array with byte 0 as the length).
The whole string is guaranteed to be printed in one sequence, two concurrent calls to str.to.screen will not interleave.
Equivalent to the program fragment :-

```
PROC str.to.screen (VALUE s []) =  
  SEQ n = [1 for s [BYTE 0]]  
  screen ! s [BYTE n] :
```

EXTERNAL PROC num.to.screen (VALUE n) :

Output a number to the screen. The number can be signed, and uses the minimum number of characters (no leading spaces). Equivalent to the 'C' language 'printf ("%d",n);' statement.

EXTERNAL PROC str.to.chan (CHAN c,VALUE s []) :

Output the string s to a channel 'c'. The call 'str.to.chan (screen,"fred")' is identical to 'str.to.screen (fred)'. Useful for string output to files.

EXTERNAL PROC num.to.chan (CHAN c,VALUE n) :

Output ascii string for the number 'n' to channel 'c'. Like 'str.to.chan' but for numbers not channels.

EXTERNAL PROC num.to.screen.f (VALUE n,d) :

Output a number to the screen in a field of width 'd'. If the number is too big for the field the number is written out in full regardless, the routine call num.to.screen.f (n,1) is equivalent to num.to.screen (n). The routine uses the 'C' language printf format %nd where n is the field width.

EXTERNAL PROC goto.x.y (VALUE x,y) :

Use the 'curses' package to implement a cursor 'goto' facility. No error checking is made that the move is within the screen area. The x-axis is across the screen and y-axis down, co-ordinate (0,0) is in the top left hand corner of the screen. The first line is used by the run time system to print messages.

EXTERNAL PROC clear.screen :

Use curses to clear the screen, if cursor addressable option not used this will still try to clear the screen using the curses "CL" termcap defined string.

EXTERNAL PROC num.from.keyboard (VAR n) :

Read a number from the keyboard and assign to variable 'n'. The routine is not very sophisticated. It will read negative numbers (start '-') and ignore any leading 'space' characters. The number must be followed by a non-digit, this character is read by the routine and not available on a subsequent 'Keyboard ? ch' process. There is no check that the number is too big for the number range. It will expect at least one digit otherwise it will give an error message.

EXTERNAL PROC num.from.chan (CHAN c, VAR n) :

Read a number from a channel 'c'. If 'c' is the keyboard this is equivalent to calling 'num.from.keyboard'.

EXTERNAL PROC abort.program :

Force the program to abort execution. An explanatory message is printed so that the cause will be known.

EXTERNAL PROC force.break :

Perform the same action as if 'CTRL-C' was pressed at the terminal. The user interface routines can then be run under the menu selection facility provided.

EXTERNAL PROC random (VALUE d, VAR n) :

Return a pseudo random number in the range 0 to d-1 by using the 'C' 'random()' function in the variable n. The VALUE of d must not be zero. The sequence of random numbers will be modified if the '-R' run option is used.

EXTERNAL PROC init.random (VALUE n) :

Initialise the seed for the random number generator for subsequent calls to the procedure 'random'. Uses the 'C' language routine 'srandom()'.

EXTERNAL PROC trace.value (VALUE n) :

Print out the integer value of 'n' on the screen with the prefix string 'Trace value : ' - this makes debugging a little easier.

EXTERNAL PROC open.file (VALUE path.name [], access [], CHAN io.chan) :

Connect the channel 'io.chan' to a UNIX file. The procedure must be provided with the pathname of the file as a string, and the access mode ("r" read access, "w" write access, "a" append access). Subsequent input or output on 'io.chan' will fetch/put a single character from/to the file. Attempts to input past the end of file will receive the value -1.

EXTERNAL PROC close.file (CHAN io.chan) :

Cease connection of the channel with its currently open file.

EXTERNAL PROC open.pipe (VALUE command.name [], access [], CHAN io.chan) :

Connect the channel 'io.chan' to a UNIX pipe running command 'command.name'. The procedure must be provided with the UNIX command name and 'r' to read from it, or 'w' to write to it). Subsequent input or output on 'io.chan' will fetch/put a single character from/to the file. Attempts to input past the end of file will receive the value -1.

EXTERNAL PROC close.pipe (CHAN io.chan) :

Cease connection of the channel with its currently active command.

EXTERNAL PROC system.call (VALUE command [], VAR code) :

Execute the UNIX command contained in the string 'command' and return the value in 'code' TRUE if the command succeeded without error and FALSE otherwise.

EXTERNAL PROC set.timers (VALUE init.value) :

Set up the interval timers ITIMER_REAL, ITIMER_VIRTUAL to the given start value. These are used for timing sections of code on the VAX. Uses 'setitimer' call. Note that using 'WAIT' primitive will reset the timer so it can only be used for simple sections of code. It should also be noted that it times the whole program and not a single occam process.

EXTERNAL PROC get.real.timer (VAR secs, micro.secs) :

Get the current elapsed timer values in seconds and microseconds. Timers count downwards and are not especially accurate. Uses 'getitimer' call.

EXTERNAL PROC get.cpu.timer (VAR secs, micro.secs) :

Get the current executed CPU timer values in seconds and microseconds. Timers count downwards and are not especially accurate.

Floating Point Library

Routines to perform floating point input/output. They are available by giving the compiler flag '-F' when linking an occam program.

Floating point value can be assigned and transmitted via channels just like normal integer values, see the file 'occamversion' for details as to the language extensions introduced to support them.

Input/Output Routines

EXTERNAL PROC fp.num.to.screen (VALUE FLOAT f) :

Print out the floating point number in 'C' language float format "%6.6f". If the number is too small or too big the standard 'C' action will be taken.

EXTERNAL PROC fp.num.to.screen.f (VALUE FLOAT f, VALUE w, d) :

Print out the floating point number in 'C' real format "%w.df". If the number is too small or too big problems will arise.

EXTERNAL PROC fp.num.to.screen.g (VALUE FLOAT f) :

Print out the floating point number in 'C' real format "%g". This will use the most appropriate format - exponent form if necessary.

EXTERNAL PROC fp.num.to.chan (CHAN c, VALUE FLOAT f) :

Write a number to a channel. If channel is 'screen' this is equivalent to 'fp.num.to.screen'. Useful for writing data to files.

EXTERNAL PROC fp.num.from.keyboard (VAR FLOAT f) :

Read in a floating point number. The number is expected to begin with a digit or '.' (indicating 0.), leading spaces are ignored. The number ends on a non-digit and this character will not be available to subsequent reads from the keyboard channel. The following are valid input numbers followed by the interpreted value for the input.

45.35 (45.35) 0.0004 (0.0004) .0 (0.0) 1. (1.0) 124 (124.0)

EXTERNAL PROC fp.num.from.chan (CHAN c, VAR FLOAT f) :

Read a floating point number from a channel 'c'. If channel is keyboard this is equivalent to 'fp.num.from.keyboard'.

Mathematical Routine Library

Mathematical routines from the UNIX '-lm' library. These are included by specifying the '-M' flag. They are all in single precision even though double precision 'C' routines are called.

EXTERNAL PROC fp.sine (VALUE FLOAT a, VAR FLOAT res) :

Return the sine of 'a' in 'res'. Angles are in radians.

EXTERNAL PROC fp.cosine (VALUE FLOAT a, VAR FLOAT res) :

Return the cosine of 'a' in 'res'. Angles are in radians.

EXTERNAL PROC fp.arc.sine (VALUE FLOAT a, VAR FLOAT res) :

Return the arc sine of 'a' in 'res'. Angles are in radians.

EXTERNAL PROC fp.arc.cosine (VALUE FLOAT a, VAR FLOAT res) :

Return the arc cosine of 'a' in 'res'. Angles are in radians.

EXTERNAL PROC fp.arc.tan (VALUE FLOAT a, VAR FLOAT res) :

Return the arc tangent of 'a' in 'res'. Angles are in radians.

EXTERNAL PROC fp.exp (VALUE FLOAT a, VAR FLOAT res) :

Return e to the power 'a' in 'res'.

EXTERNAL PROC fp.log (VALUE FLOAT a, VAR FLOAT res) :

Natural logarithm of 'a' in 'res'.

EXTERNAL PROC fp.sqrt (VALUE FLOAT a, VAR FLOAT res) :

Square root of 'a' in 'res'. Returns an occam error if 'a' is negative.

The run time system

As you might hope when an occam program is executed it will follow the program execution until one of three things happen.

- 1] The program terminates
- 2] CTRL-C is pressed on the keyboard
- 3] An error is detected.

In the case of (2) and (3) a debug option will be displayed, this allows you to abort the program, ignore the interrupt (continue), and to restart the program again. Other options control the '-e' trace output, provide a 'system' debug option (which is only really useful to someone who knows their way around the compiler), an option to specify which source file you want to debug and the 'screen animated debug'. This later option should be of most use and is described in detail in the next section.

Errors come in two types 'Fatal Errors' and just 'Errors', it is not possible (or wise) to continue execution after the former, but the latter may be ignored if the symptom is expected.

The run time display debugger

This utility that runs under the run time system enables users to look at the status of the processes during execution of a program.

The utility requires the use of a cursor addressable terminal. The system provides selective display of the source file(s) that were compiled to form the program together with a column showing the currently existing processes on those particular lines of the source file.

When initially entered by pressing 'CTRL-C' the program execution will be halted, the execution can be restarted in 'stepped mode' so that the display will be updated every occam scheduler action.

Breakpoints and trace points can be added at selected line numbers. Break points cause the debug display to be automatically entered when any of the process executes any of the source lines on which a break point is set. Trace points cause temporary entry into the debug display before resuming normal execution after five seconds pause.

If a file has been compiled with the '-G' flag then the value of occam variables and the status of channels can be printed. Because an occam program can have several processes running with different values to the same identifiers (e.g. within PAR n = [0 FOR 7], 'n' has a different value for each separate process) a single process must be selected as before this facility can be used. When selected a second window within the debug display is opened and the values printed by the program are placed within it.

Straightforward use of the debug display will normally entail running a program and pressing CTRL-C when a dubious section of code is about to be executed and entering the debug display ('z' command). Thereafter the commands 'p' to find the next process, 'i' and 'b' might be used to see whereabouts the process is executing. The program can then be single stepped through using the 'r' command to start execution and 's' command to stop execution. Eventually exit of the debug display can be made with the 'x' command.

There are two special markers that are used, '>' on a line indicates the currently selected line and '-' the currently selected process.

The commands where practical have been made similar to those in UNIX 'vi'. (UNIX is a trademark of A.T. & T.).

Available commands

Moving about within the file

- !D- Move forward half a page of source text.
- !P- Move forward a page of source text.
- !U- Move backward half a page of source text.
- !B- Move backward a page of source text.
- :<number> - Move to given line <number> in file.
- k - (or !K) Move down one line.
- j - (or !J) Move up one line.
- /<string> - Find given <string> in file from current position.
- n - Find next string occurrence for match string selected by '/' command.
- p - Find the next process in the file.

Trace/Breakpoints

- b - Add breakpoint at currently selected line.
- t - Add tracepoint at currently selected line.
- d - Delete the trace/break point at the selected line.
- c - Delete all the points in the current file.
- C - Delete all the points in all the files.
- P - Print process status of the currently selected process
- D - Deselect the current debug occam process.
- S - Select the current debug occam process.
- N - Select next process on the same line, if there are several processes that are shown as executing on the same line then 'S' will make an arbitrary choice, 'N' can be used to override this and step through the processes until the one that is desired is selected.

Symbol inspection

- m - Select a symbol to display, if no symbols have been selected before then the symbol window is opened and the value of the variable or the status of a channel.
- M - Repeat the previous 'm' command. To find the value of the same variable name again.

Execution control

- a - Abort the run.
- r - Run debug display if a debug process is selected the debug display will be re-entered every time that process is run, otherwise the debug display will be run each time any process is run.
- > - Execute in single step mode. Only a single step is executed.
- s - Stop the debug display from running temporarily after a 'r' or 'x' command.
- u - Change display step interval (initial step interval is 1), this permits the location of processes to be seen after 'n' steps rather than after each and every time it is executed. Not particularly useful.
- x - Exit display debugger, program will proceed normally until a trace/break point is found or 'tC' is pressed.
- X - Exit to main '' menu so that program restart, abort, file selection or system debug can be done. Used when you wish to debug a different file or to set things going again after setting up breakpoints.

Miscellaneous

- ? - Print out this help information.
- +L- (or +R) Redraw the current displayed information.
- i - Buffer keyboard channel input text for the program.
- O - Print overall data about the processes currently executing - how many are in each process status, stack use and clock time.
- V - Display the occam program's current screen output temporarily
- v - Invoke the 'view' command on the occam source file (this is just like 'vi' but with read only access to the file - This can be used to provide more powerful string search facilities when debugging).

Display key

The column between the line number and the text is used to display the number and status of processes executing on that line. Because of the compilation these may be out by a line or two in some circumstances. Most sequential code will be executed as a single block - so a process will not move through a SEQ block one step at a time necessarily.

The special symbol 'P' does not represent a process, it indicates that a procedure has been called at that point. 'P' therefore represents the 'call point' of the procedure.

The following symbols are used to represent the various process stati :-

- * - An active process - may be chosen for execution at any time.
- a - Process waiting for one or more ALT guards to become TRUE.
- w - Process waiting for a clock time or for input/output.
- c - Process is waiting for one or more child PAR processes to terminate.

In addition break and trace points are indicated in the column by giving a 'T' for a trace point and 'B' for a break point.

So a display of :-

```
316:3*w      : occam.s ? razor
```

Indicates that there are three active processes and one process waiting input on line 316.

Keyboard and Screen input/output

Because the debug display routine is fully interactive the screen and keyboard data from the program can not be handled in the same manner as normal. Input for the keyboard must be input using the 'i' command - a whole line can be input and will be buffered up for program input in this way. Screen output should be displayed as it is produced (but a copy of it will be sent to the screen image that will redisplayed on exit from the display debugger) or the 'v' command. Strings can have escapes in them '*n' means newline, '*r' carriage return and '*' space.

Non standard occam features

This compiler to the best of my knowledge (Mr.R.P. Stallard of the Department of Computer Studies, Loughborough University of Technology, U.K.) implements the occam language as defined in the occam programming manual published by INMOS limited subject to a few restrictions and extensions that are described in this file. These differences are intended to make transfer of occam programs from different implementations feasible.

It is intended to be compatible to the INMOS booklet version and the Prentice Hall book definition. OCCAM, INMOS and Transputer are registered trademarks of the INMOS Group of Companies.

INMOS proto-occam language revisions

The following additional features introduced into INMOS occam products can now be selected by the compiler flag option '-I'.

STOP primitive.
TIME channel.
IF on finding none of the conditions TRUE STOPS.

Restrictions

These restrictions are either optional features as described in the published language definition or compiler restrictions unlikely to limit ordinary use of occam.

No configuration section rules.
The operator '>>>' uses VAX shift right operator.
No prioritized PAR, all parallel processes have equal priority.
Number of arguments to a procedure limited to 255 maximum.
AFTER returns a time difference not a boolean value.

Extensions

PAR replicator count and base can be variables
A variable number of processes can be created by replicated PAR.

Recursive calls to procedures permitted
A procedure can call itself.

Screen channel can be used by more than one process
The special screen channel can be accessed by any number of different occam processes. This facilitates debugging of occam programs and is not difficult to implement.

Multiple source file compilation

Procedures and Variables can be defined in one file and referenced in another.

The definition is preceded by the new keyword 'LIBRARY' before 'PROC' and the definition must be at the outer level of program nesting.

References to procedures in other files are defined by preceding 'PROC' by 'EXTERNAL' and replacing the '=' start of procedure definition by ':' to indicate end of definition.

e.g.
File main.occ File sub.occ

EXTERNAL PROC f (value n) : LIBRARY PROC f (value n) =
SEQ SEQ
 f (27) num.to.screen (a*102)
 str.to.screen ("Enter next"):

The two files can be compiled by :-

occam main.occ sub.occ to compile both together
occam sub.occ -l to compile sub.occ separately
occam main.occ sub.o to link in the pre-compiled sub.occ file

IN 3.0 THIS HAS BEEN extended to variables and channels, in the case of vectors of variables and channels the size need not be specified but the type must be :-

Defining file :-

LIBRARY CHAN network,comms [56] :
LIBRARY VAR blot [BYTE 4],spot [42] :
LIBRARY VAR FLOAT hyper,bolic [2],active [17] :

Referring file :-

EXTERNAL CHAN network,comms [] :
EXTERNAL VAR blot [BYTE],spot [],bolic [FLOAT] :
EXTERNAL VAR FLOAT hyper,active [] :

Floating point arithmetic

The compiler permits the use of floating point numbers and arithmetic operators. The compiler uses 32 bit VAX floating point throughout.

Floating point numbers are declared by following VAR by the new keyword float :-

VAR FLOAT x,y,factor : - Floating point number declaration
VAR num,ply : - Normal occam variables.

Floating point number constants are supported these may be in two forms with decimal point or with decimal point and exponent :-

```
x := 1.45
y := 2.3e-23 + 3.4e+1  -- Note that the exponent must be given a sign
```

The following operators may be used on floating point numbers (both operands must be floating point)

+ - * / < > <= >= = <> - (monadic minus)

```
x := 1.3 + (y * factor)
```

```
IF
```

```
  x > 67.8
```

```
  y := -3.4          -- Note use of monadic minus.
```

Parameters to procedures must also have type set to VAR FLOAT or VALUE FLOAT - the actual parameters must be of the same type.

```
PROC sum (VALUE FLOAT a [], b [], VAR FLOAT res [], VALUE n) =
```

```
  PAR i = [0 FOR n]
```

```
    res [i] := a [i] + b [i] :
```

```
  VAR FLOAT t [23], s [45], w [32] :
```

```
  --
```

```
  sum (t, s, w, i2)
```

Floating values may be transmitted along channels - but there are no checks that the sender and receiver both expect floating point values.

Input of floating point numbers can be carried out by calling the library routine 'fp.num.from.keyboard' and output by the routine 'fp.num.to.screen'.

Interconversion of floating point and integers is performed by the assignment operator :-

```
num := x  -- Convert floating 'x' to integer 'num'
```

```
y := num  -- Convert integer 'num' to floating 'y'
```

Attempts to use logical and shift operators on floating point numbers are flagged as errors.

Appendix D

SELECTED SYSTOLIC PROGRAMS

[illegible]

```

66:      TRUE
67:      --      Output a dummy result for the flowing array
68:      r.out ! FALSE
69:      --      Calculation
70:      PAR
71:      count := count+1
72:      s[1] := s[0]
73:      p[1] := p[0]
74:      SEQ
75:      r := r AND ( s[0] = p [0] )
76:      if
77:      ip = ipselect
78:      put ( FALSE , " Error from ips number ip " ) :
79:  --
80:  --***** FLOW for the results *****
81:  --
82:  -- This an array structure to flow the correct sequencing of the
83:  -- comparison results out of the pattern matcher array.
84:  --
85:  PROC flow ( Chan r.in, f.in, f.out, ctl.in ) =
86:  VAR f[2], r, ctl :
87:  SEQ
88:  f[1] := FALSE
89:  ctl := 0
90:  WHILE ctl <> (-1)
91:  SEQ
92:  PAR
93:  --      I/O operations
94:  f.in ? f[0]
95:  r.in ? r
96:  ctl.in ? ctl
97:  f.out ! f[1]
98:  --      Calculation.
99:  f[1] := r OR f[0] :
100:  --
101:  --***** SOURCE for string definition *****
102:  --
103:  -- Alternatively this procedure outputs, every clock pulse, a
104:  -- string character or a dummy element to the right boundary
105:  -- cell in the array. A control signal is, however, required
106:  -- to be broadcasted to all the array components through the ctl
107:  -- channels instructing them to terminate processing.(Not neces-
108:  -- sary in a hard-soft systolic design.
109:  --
110:  PROC source.string ( CHAN s.out, f.out, ctl.out[] ) =
111:  VAR ch[2], alter, dummy :
112:  CHAN str.in :
113:  SEQ
114:  open.file ("text","r",str.in)
115:  dummy := '$'
116:  --      Pass over patlen / 2 characters
117:  SEQ i = [0 FOR patlen/2]
118:  str.in ? ANY
119:  IF
120:  (patlen \ 2) = 0
121:  --The first character to be sent must be DUMMY element.
122:  alter := FALSE
123:  TRUE
124:  --The first character to be sent must be the current
125:  --string character.
126:  alter := TRUE
127:  --      Read the current character in ch.
128:  str.in ? ch[1]
129:  WHILE ch[1] <> (-1)
130:  SEQ
131:  IF

```

```

132:         alter
133:         SEQ
134:         -- I/O operations.
135:         PAR
136:             str.in ? ch[0]
137:             s.out ! ch[1]
138:             f.out ! FALSE
139:             PAR i = [0 FOR 2*(patlen+1) ]
140:                 ctl.out[i] ! ch[1]
141:         -- calculations.
142:         PAR
143:             alter := FALSE
144:             ch[1] := ch[0]
145:     TRUE
146:     SEQ
147:     -- I/O operations.
148:     PAR
149:         s.out ! dummy
150:         f.out ! FALSE
151:         PAR i = [0 FOR 2*(patlen+1) ]
152:             ctl.out[i] ! 0
153:     -- calculations.
154:     alter := TRUE
155:     -- patlen+2 dummy characters are sent to the
156:     -- the array so that all the string is com-
157:     -- pletely processed.
158:     SEQ i = [ 0 FOR patlen +2 ]
159:     PAR
160:         s.out ! dummy
161:         f.out ! FALSE
162:         PAR j = [0 FOR 2*(patlen+1)]
163:             ctl.out[j] ! 0
164:     -- Now the EOF signal is broadcasted to all cells
165:     PAR
166:         s.out ! dummy
167:         f.out ! FALSE
168:         PAR j = [0 FOR 2*(patlen+1)]
169:             ctl.out[j] ! ch[1]
170:     close.file (str.in):
171:
172:     --
173:     ----- SOURCE for pattern definition -----
174:     --
175:     -- Alternatively this procedure sends, every clock cycle, a
176:     -- pattern character or a dummy element to the left boundary
177:     -- cell of the array.
178:     PROC source.pattern ( CHAN p.out, ctl.in ) =
179:     VAR ctl, dummy, alter, ind :
180:     SEQ
181:     PAR
182:         ctl := 0
183:         alter := TRUE
184:         ind := 0
185:         dummy := '$'
186:     WHILE ctl <> (-1)
187:     SEQ
188:     IF
189:         alter
190:         -- the current pattern character is to be sent.
191:     SEQ
192:         -- I/O operations
193:     PAR
194:         ctl.in ? ctl
195:         p.out ! pat[ BYTE ind+1]
196:     -- Calculation
197:     PAR

```

```

198:             ind := (ind +1)/patlen
199:             alter := FALSE
200:         TRUE
201:             --             the dummy element is to be sent.
202:         SEQ
203:             --             I/O operations
204:         PAR
205:             ctl.in ? ctl
206:             p.out ! dummy
207:             --             Calculation.
208:             alter := TRUE :
209:         --
210:         ----- SINK definition -----
211:         --
212:         --
213:         -- The sink procedure inputs a signal from the left hand flow
214:         -- cell. However, string and pattern characters output from the
215:         -- left and right boundary ips cells respectively are absorbed
216:         -- the sink mainly to avoid using other type of cells than that
217:         -- already being used. The result is analysed and and in the
218:         -- case of a success an eventual print out is performed.
219:         --
220:         PROC sink ( CHAN s.in, p.in, f.in, ctl.in ) =
221:             VAR ch, ctl, f, chpos:
222:             SEQ
223:                 PAR
224:                     chpos:= - 1
225:                     ctl := 0
226:                 WHILE ctl <> (-1)
227:                     SEQ
228:                         --             I/O operations.
229:                     PAR
230:                         s.in ? ANY
231:                         p.in ? ANY
232:                         ctl.in ? ctl
233:                         f.in ? f
234:                         chpos:=chpos+1
235:                     --             Calculations.
236:                 IF
237:                     f
238:                     put (chpos," Pattern found at " ) :
239:                 --
240:         ----- SYSTEM configuration -----
241:         --
242:         --
243:         -- The system is specified by indicating the corresponding
244:         -- channels that link sources, ips's and sink to form
245:         -- the solution network .
246:         --
247:         PROC system =
248:             PAR
249:                 source.string ( s.c[patlen], f.c[patlen], ctl.c )
250:                 source.pattern ( p.c[0], ctl.c[patlen] )
251:                 PAR i = [0 FOR patlen ]
252:                     PAR
253:                         ips (i,s.c[i+1],s.c[i],p.c[i],p.c[i+1],r.c[i],ctl.c[i])
254:                         flow ( r.c[i], f.c[i+1], f.c[i], ctl.c[ patlen+(i+2)])
255:                     sink ( s.c[0], p.c[patlen], f.c[0], ctl.c[patlen+1] ):
256:             --
257:             --
258:             --
259:             --
260:         ----- Pattern input procedure -----
261:         --
262:         PROC inp.pattern =
263:             VAR ch :

```

```

264:      SEQ
265:      --          input pattern characters
266:      str.to.screen ( " Input pattern  ")
267:      patlen:= 0
268:      keyboard ? ch
269:      WHILE ch <> '*n'
270:          SEQ
271:              patlen:= patlen+1
272:              pat[BYTE patlen] := ch
273:              screen ! pat[BYTE patlen]
274:              keyboard ? ch
275:      pat[BYTE 0]:= patlen
276:      get ( ipselect, " Input a value for the selected ips " ) :
277:
278:      --
279:      ----- string input procedure -----
280:      --
281:      --      This procedure opens the "string" file and reads the first
282:      --      (patlen / 2 ) characters and stored them in string which is
283:      --      used at the initialisation phase of every ips in the array.
284:      PROC inp.string =
285:          CHAN str.in :
286:          SEQ
287:              open.file ("text","r",str.in)
288:              SEQ i = [0 for patlen/2]
289:                  str.in ? string [ BYTE i+1]
290:              string [ BYTE 0] := patlen / 2
291:              close.file (str.in) :
292:
293:      --
294:      ----- MAIN program -----
295:      --
296:      SEQ
297:          inp.pattern
298:          inp.string
299:          system

```

```

1:      --          *** PROGRAM 7.2 ***
2:      --
3:      --
4:      --<><>+<><>+<><>+<><>+<><>+<><>+<><>+<><>+<><>
5:      --<>
6:      --<> Pattern Matcher Soft-systolic Algorithm <>
7:      --<> Model R2 ( An array of special cells<>
8:      --<> is used to flow out the <>
9:      --<> results output from all <>
10:     --<> the ips cells. ) <>
11:     --<>
12:     --<><>+<><>+<><>+<><>+<><>+<><>+<><>+<><>+<><>
13:     --
14:     --The string characters, si' s, and the pattern characters,
15:     --pi' is move systolically in the same direction but at diffe-
16:     --rent speed; Si' s move as twice as fast as pi' s.
17:     --The results stay in the cells.
18:     EXTERNAL PROC open.file ( VALUE path.name [], access [],
                                CHAN io.chan ) :
19:     EXTERNAL PROC close.file ( CHAN io.chan ) :
20:     EXTERNAL PROC put ( VALUE n, s[] ) :
21:     EXTERNAL PROC get ( VAR v, VALUE s[] ) :
22:     EXTERNAL PROC get.n ( VAR v[], VALUE n, s[] ) :
23:     EXTERNAL PROC str.to.screen ( VALUE s[] ) :
24:     --
25:     -- Define maximum pattern length.
26:     DEF mo = 15:
27:     -- Declare pattern and string storage.
28:     VAR pat [ BYTE mo ] :
29:     -- Actual parameters.
30:     VAR patlen :
31:     -- Define all the system channels.
32:     CHAN s.c[mo+1], p.c[mo+1], r.c[mo], f.c[mo+1], ctl.c[(2*mo)+1] :
33:     --
34:     --***** IPS cell definition *****
35:     --
36:     PROC ips ( VALUE ip,CHAN s.in,s.out,p.in,p.out,r.out,ctl.in )=
37:         VAR s[2], p[2], r, count, ctl :
38:         SEQ
39:             -- Initialisation
40:             PAR
41:                 count := patlen-ip
42:                 s[1] := 0
43:                 p[1] := 0
44:                 r := FALSE
45:                 ctl := 0
46:             WHILE ctl <> (-1)
47:                 SEQ
48:                     -- Input/output operations.
49:                     PAR
50:                         s.in ? s[0]
51:                         p.in ? p[0]
52:                         s.out ! s[1]
53:                         p.out ! p[1]
54:                         ctl.in ? ctl
55:                     IF
56:                         (count \ patlen ) = 0
57:                         SEQ
58:                             r.out ! r
59:                             r := TRUE
60:                         -- output a dummy result to be used by the
61:                         -- flowing array.
62:                         TRUE
63:                             r.out ! FALSE
64:                     -- Calculation operation
65:                     PAR

```

```

66:          s[1] := s [0]
67:          p[1] := p [0]
68:          count := count+1
69:          r := r AND (s[0] = p[0]) :
70:
71:  --
72:  --***** DELAY definition *****
73:  --
74:  -- This procedure delays the patern character stream by a
75:  -- single cycle
76:  PROC delay ( CHAN p.in, p.out, ctl.in ) =
77:      VAR p[2], ctl :
78:      SEQ
79:          ctl := 0
80:          p[1] := 0
81:          WHILE ctl <> (-1)
82:              SEQ
83:                  -- I/O operation.
84:                  PAR
85:                      p.in ? p[0]
86:                      ctl.in ? ctl
87:                      p.out ! p[1]
88:                  -- calculation
89:                  p[1] := p[0] :
90:
91:  --
92:  --***** FLOW for results *****
93:  --
94:  -- This procedure is a cell helping to flow out results
95:  -- of the pattern matcher array in a correct sequencing.
96:  --
97:  PROC flow ( CHAN r.in, f.in, f.out, ctl.in ) =
98:      VAR f[2], r, ctl :
99:      SEQ
100:          f[1] := FALSE
101:          ctl := 0
102:          WHILE ctl <> (-1)
103:              SEQ
104:                  PAR
105:                      -- I/O opeation
106:                      f.in ? f[0]
107:                      r.in ? r
108:                      ctl.in ? ctl
109:                      f.out ! f[1]
110:                      -- Calculation
111:                      f[1] := f[0] OR r :
112:
113:  --
114:  --***** SOURCE definition *****
115:  --
116:  -- This procedure outputs every clock pulse a string character
117:  -- and a true signal ,used in the result, to the left boundary
118:  -- cell in the array. A control signal is, however, required
119:  -- be broadcasted to all the array components ( ctl channel
120:  -- array ) to terminate processing. ( In a hard design this
121:  -- is not necessary).
122:  --
123:  PROC source ( CHAN s.out, p.out, f.out, ctl.out[] ) =
124:      VAR ch, ind :
125:      CHAN str.in :
126:      SEQ
127:          -- input string characters
128:          ch := 0
129:          ind := 0
130:          open.file ("text","r",str.in)
131:          WHILE ch <> (-1)

```

```

132:      SEQ
133:      --                                I/O operations.
134:      PAR
135:          str.in ? ch
136:          p.out ! pat[ BYTE ind+1]
137:          f.out ! FALSE
138:      --                                calculations.
139:      PAR
140:          s.out ! ch
141:          ind := (ind+1)/patlen
142:          PAR i = [0 for (2*patlen)+1 ]
143:              ctl.out[i] ! ch
144:      close.file (str.in):
145:
146:      --
147:      ----- SINK definition -----
148:      --
149:      --
150:      --The sink procedure inputs two data, a string character from
151:      --the right end delay cell, a result from the right end ips and
152:      --a control signal from the source. The result is analysed and
153:      --and in a successful case an eventual print out is performed.
154:      --
155:      PROC sink ( CHAN s.in, p.in, f.in, ctl.in) =
156:          VAR ch, ctl, f, chpos:
157:          SEQ
158:              chpos:= -(2*patlen)
159:              ctl := 0
160:              WHILE ctl <> (-1)
161:                  SEQ
162:                      --                                I/O operations.
163:                      PAR
164:                          s.in ? ANY
165:                          p.in ? ANY
166:                          ctl.in ? ctl
167:                          f.in ? f
168:                          chpos:=chpos+1
169:                      --                                Calculations.
170:                      IF
171:                          f
172:                          put (chpos," Pattern found at " ) :
173:      --
174:      ----- SYSTEM configuration -----
175:      --
176:      --    The system is specified by indicating the corresponding
177:      --    channels that link source, ips's ,delay and sink to form
178:      --    the solution network .
179:      --
180:      PROC system =
181:          PAR
182:              source ( s.c[0], p.c[0], f.c[0], ctl.c )
183:              PAR i = [0 FOR patlen ]
184:                  PAR
185:                      ips ( i,s.c[i],s.c[i+1],p.c[2*i],p.c[(2*i)+1],r.c[i],
186:                          ctl.c[(2*i)+1] )
187:                      delay ( p.c[(2*i)+1], p.c[2*(i+1)], ctl.c[(2*i)+1] )
188:                      flow (r.c [i],f.c[i],f.c[i+1], ctl.c[(2*patlen)+(i+2)])
189:                      sink ( s.c[patlen],p.c[2*patlen], f.c[patlen],
190:                          ctl.c[(2*patlen)+1] ):
191:      --
192:      --
193:      ----- Pattern input procedure -----
194:      --
195:      PROC inp.pattern =

```

```
196:      VAR ch :
197:      SEQ
198:      --          input pattern characters
199:      str.to.screen ( " Input pattern ")
200:      patlen:= 0
201:      keyboard ? ch
202:      WHILE ch <> '*n'
203:          SEQ
204:              patlen:= patlen+1
205:              pat[BYTE patlen] := ch
206:              screen ! pat[BYTE patlen]
207:              keyboard ? ch
208:      pat[BYTE 0]:= patlen :
209:
210:      --
211:      --***** MAIN program *****
212:      --
213:      SEQ
214:      inp.pattern
215:      system
```

[illegible]

```

66:      SEQ
67:      --          input string characters
68:      open.file ("text","r",str.in)
69:      ch := 0
70:      WHILE ch <> (-1)
71:          SEQ
72:          --          I/O operations.
73:          str.in ? ch
74:          PAR
75:              PAR j = [ 0 FOR (patlen+1) ]
76:                  s.out [j] ! ch
77:              --          Output the test accumulator.
78:              r.out ! TRUE
79:      close.file (str.in):
80:      --
81:      --***** SINK definition *****
82:      --
83:      --
84:      -- Gets as input a matching result from the right end cell and
85:      -- depending the success of the search, outputs the pattern
86:      -- position in the string.
87:      --
88:      PROC sink ( CHAN s.in, r.in ) =
89:          VAR ch, r, chpos:
90:          SEQ
91:              chpos:= -patlen
92:              ch := 0
93:              WHILE ch <> (-1)
94:                  SEQ
95:                  --          I/O operations.
96:                  PAR
97:                      s.in ? ch
98:                      r.in ? r
99:                      chpos:=chpos+1
100:                  --          Calculations.
101:                  IF
102:                      r
103:                      put (chpos," Pattern found at " ) :
104:      --
105:      --***** SYSTEM configuration *****
106:      --
107:      -- The system procedure identifies all the connecting channels
108:      -- That link all the array components in order to form the
109:      -- required solution network.
110:      --
111:      PROC system =
112:          PAR
113:              source.string ( s.c, r.c[0] )
114:              PAR i = [0 FOR patlen ]
115:                  ips ( i, s.c[i], r.c[i], r.c[i+1] )
116:              sink ( s.c[patlen], r.c[patlen] ):
117:      --
118:      --***** Patern input procedure *****
119:      --
120:      --
121:      PROC inp.pattern =
122:          VAR ch :
123:          SEQ
124:              --          input pattern characters
125:              str.to.screen ( " Input pattern " )
126:              patlen:= 0
127:              keyboard ? ch
128:              WHILE ch <> '*n'
129:                  SEQ
130:                      patlen:= patlen+1
131:                      pat[BYTE patlen] := ch

```

```
132:         screen ! pat[BYTE patlen]
133:         keyboard ? ch
134:         pat[BYTE 0]:= patlen :
135:
136:         --
137:         --***** MAIN program *****
138:         --
139:         SEQ
140:         inp.pattern
141:         system
```

```

1:      ---                                     *** PROGRAM 7.4 ***                                465
2:      ---
3:      ---
4:      --<><>+<><>+<><>+<><>+<><>+<><>+<><>+<><>+<><>
5:      ---
6:      --          Pattern Matcher Soft-systolic Algorithm
7:      --                      Model B2 ( Broadcasting si' s )
8:      ---
9:      --<><>+<><>+<><>+<><>+<><>+<><>+<><>+<><>+<><>
10:     ---
11:     --    si' s, the input string characters, are broadcasted to all
12:     --    ips ( Comparator and accumulator cell ), ri' s, the tempora-
13:     --    lly stored result of a partial comparison, are output cycli-
14:     --    cally one at a time and, the pattern characters, pi' s move
15:     --    cyclically from left to righth.
16:     ---
17:     EXTERNAL PROC open.file ( VALUE path.name [], access [],
                               CHAN io.chan ) :
18:     EXTERNAL PROC close.file ( CHAN io.chan ) :
19:     EXTERNAL PROC put ( VALUE n, s[] ) :
20:     EXTERNAL PROC get ( VAR v, VALUE s[] ) :
21:     EXTERNAL PROC get.n ( VAR v[], VALUE n, s[] ) :
22:     EXTERNAL PROC str.to.screen ( VALUE s[] ) :
23:     ---
24:     --          Define maximum pattern length.
25:     DEF mo = 15:
26:     --          Declare pattern storage.
27:     VAR pat [ BYTE mo ] :
28:     --          Actual parameters.
29:     VAR patlen :
30:     --          Define all the system channels.
31:     CHAN s.c [mo+1], r.c [mo+1], p.c[mo+1] :
32:     ---
33:     -----***** IPS cell definition -----*****
34:     ---
35:     PROC ips ( VALUE ip, CHAN s.in, p.in, p.out, r.out )=
36:         VAR p[2], r, ch, count :
37:         SEQ
38:             --          Global initializations.
39:             PAR
40:                 p[0] := 0
41:                 r := FALSE
42:                 p[1] := pat[ BYTE (patlen-ip)]
43:                 ch := 0
44:                 count := patlen - ip
45:             WHILE ch <> (-1)
46:                 SEQ
47:                     --          Input/output operations.
48:                     PAR
49:                         s.in ? ch
50:                         p.in ? p[0]
51:                         p.out ! p[1]
52:                     IF
53:                         (count \ patlen ) = 0
54:                         SEQ
55:                             r.out ! r
56:                             r := TRUE
57:                     --          Calculation
58:                     PAR
59:                         count := count + 1
60:                         p[1] := p[0]
61:                         r := r AND ( ch = p[0]):
62:
63:     ---
64:     -----***** SOURCE definition -----*****
65:     ---

```

```

66:  --
67:  -- Broadcasts, character per character, a string of character
68:  -- to all the ips cells and the sink also.
69:  --
70:  PROC source.string ( CHAN s.out[] ) =
71:      VAR ch :
72:      CHAN str.in :
73:      SEQ
74:          --          input string characters
75:          open.file ("text","r",str.in)
76:          ch := 0
77:          WHILE ch <> (-1)
78:              SEQ
79:                  --          I/O operations.
80:                  str.in ? ch
81:                  PAR j = [ 0 FOR (patlen+1) ]
82:                      s.out [j] ! ch
83:                  close.file (str.in):
84:  --
85:  --***** SINK definition *****
86:  --
87:  --
88:  -- Gets from the input channels "TRUE" and "FALSE" signals
89:  -- outputs to the screen the position of the pattern in the
90:  -- string every time it gets a "TRUE" signal.
91:  --
92:  PROC sink ( CHAN s.in, r.in[] ) =
93:      VAR ch, r, chpos:
94:      SEQ
95:          chpos := -patlen
96:          ch := 0
97:          WHILE ch <> (-1)
98:              SEQ
99:                  --          I/O operations.
100:                  PAR
101:                      s.in ? ch
102:                      chpos:=chpos+1
103:                      ALT i = [ 0 FOR patlen]
104:                          r.in[i] ? r
105:                          SKIP
106:                  --          Calculations.
107:                  IF
108:                      r
109:                      SEQ
110:                          put (chpos," Pattern found at " ) :
111:  --
112:  --***** SYSTEM configuration *****
113:  --
114:  -- The system is specified by indicating the corresponding
115:  -- channels linking source, ips' s and the sink in a network.
116:  --
117:  PROC system =
118:      VAR ind :
119:      SEQ
120:          ind := patlen-1
121:          PAR
122:              source.string ( s.c )
123:              PAR i = [0 FOR ind]
124:                  ips ( i, s.c[i], p.c[i], p.c[i+1], r.c[i] )
125:              --          cell linkage to form the loop.
126:              ips ( ind, s.c[ind], p.c[ind], p.c[0], r.c[ind] )
127:              sink ( s.c[patlen], r.c ):
128:  --
129:  --
130:  --
131:  --

```

```

132:  --***** Pattern input procedure *****
133:  --
134:  PROC inp.pattern =
135:      VAR ch :
136:      SEQ
137:          --          input pattern characters
138:          str.to.screen ( " Input pattern  ")
139:          patlen:= 0
140:          keyboard ? ch
141:          WHILE ch <> '*'
142:              SEQ
143:                  patlen:= patlen+1
144:                  pat[BYTE patlen] := ch
145:                  screen ! pat[BYTE patlen]
146:                  keyboard ? ch
147:                  pat[BYTE 0]:= patlen :
148:
149:  --
150:  --***** MAIN program *****
151:  --
152:  SEQ
153:      inp.pattern
154:      system

```



```

66:  -- This procedure broadcasts every cycle a character from the
67:  -- input string to all the ips cells in the array. Also a contr-
68:  -- ol signal is required to be broadcasted to the array elements
69:  -- instructing them when to terminate processing. ( this is not
70:  -- necessary for the hardware implementation of this design.)
71:  --
72:  PROC source ( CHAN s.out, ctl.out[] ) =
73:      VAR ch :
74:      CHAN str.in :
75:      SEQ
76:          --          input string characters
77:          ch := 0
78:          open.file ("text","r",str.in)
79:          SEQ i = [ 0 FOR patlen -1]
80:              str.in ? ANY
81:          WHILE ch <> (-1)
82:              SEQ
83:                  --          I/O operations.
84:                  str.in ? ch
85:                  --          calculations.
86:                  PAR
87:                      s.out ! ch
88:                      PAR i = [0 for patlen + 2]
89:                          ctl.out[i] ! ch
90:                  close.file (str.in):
91:
92:  --
93:  ---***** ADDER cell definition *****
94:  --
95:  --Gets as input all the single character comparison results from
96:  --every ips in the array. These are fanned-in and summed up in
97:  --this procedure.
98:  --
99:  PROC adder (CHAN r.in[], r.out, ctl.in ) =
100:      VAR ctl, r[mo], res :
101:      SEQ
102:          ctl := 0
103:          res := FALSE
104:          WHILE ctl <> (-1)
105:              SEQ
106:                  --          I/O operations
107:                  PAR
108:                      ctl.in ? ctl
109:                      PAR i = [0 FOR patlen]
110:                          r.in[i] ? r[i]
111:                          r.out ! res
112:                  --          calculation.
113:                  SEQ
114:                      res := TRUE
115:                      SEQ i = [0 FOR patlen]
116:                          res := res AND r[i] :
117:
118:  --
119:  ---***** SINK definition *****
120:  --
121:  --
122:  -- This procedure which gets the result output from the ADDER
123:  -- prints the actual location in the string in the case of a
124:  -- pattern match.
125:  PROC sink ( CHAN s.in, r.in, ctl.in) =
126:      VAR ctl, r, chpos:
127:      SEQ
128:          chpos:= -2
129:          ctl := 0
130:          WHILE ctl <> (-1)
131:              SEQ

```

```

132:          --          I/O operations.
133:          PAR
134:              ctl.in ? ctl
135:              r.in? r
136:              s.in ? ANY
137:              chpos:=chpos+1
138:          --          Calculations.
139:          IF
140:              r
141:              put (chpos," Pattern found at " ) :
142:          --
143:          ----- SYSTEM configuration -----
144:          --
145:          -- The system is specified by indicating the corresponding
146:          -- channels that link source, ips's ,adder and sink to form
147:          -- the solution network .
148:          --
149:          PROC system =
150:              PAR
151:                  source ( s.c[0], ctl.c )
152:                  PAR i = [0 FOR patlen ]
153:                      ips ( i, s.c[i], s.c[i+1], r.c[i], ctl.c[i] )
154:                  adder ( r.c, r.c[patlen], ctl.c[patlen] )
155:                  sink ( s.c[patlen], r.c[patlen], ctl.c[patlen+1] ):
156:              --
157:          ----- Pattern input procedure -----
158:          --
159:          PROC inp.pattern =
160:              VAR ch :
161:              SEQ
162:                  --          input pattern characters
163:                  str.to.screen ( " Input pattern " )
164:                  patlen:= 0
165:                  keyboard ? ch
166:                  WHILE ch <> '*n'
167:                      SEQ
168:                          patlen:= patlen+1
169:                          pat[BYTE patlen] := ch
170:                          screen ! pat[BYTE patlen]
171:                          keyboard ? ch
172:                  pat[BYTE 0]:= patlen :
173:              --
174:          ----- string input procedure -----
175:          --
176:          -- This procedure open the string file and read the first patlen
177:          --
178:          -- characters in str which is used at the initialisation phase
179:          -- of the ips's.
180:          PROC inp.string =
181:              CHAN str.in :
182:              SEQ
183:                  open.file ("text","r",str.in)
184:                  SEQ i = [0 for patlen-1]
185:                      str.in ? str [ BYTE i+1]
186:                  str[ BYTE 0 ] := patlen-1
187:                  close.file (str.in) :
188:              --
189:          ----- MAIN program -----
190:          --
191:          --
192:          SEQ
193:              inp.pattern
194:              inp.string
195:              system
196:

```

[illegible]

```

66:
67:  --
68:  ---***** SOURCE definition *****
69:  --
70:  --This procedure outputs a string character to the left boundary
71:  --cell in the array. A control signal is, however, required to b

72:  --broadcasted to the fan-in and the sink ( ctl channel array )
73:  --to terminate processing. ( In a hard design this is not
74:  --necessary).
75:  --
76:  PROC source.string ( CHAN s.out, ctl.out[] ) =
77:    VAR ch :
78:    CHAN str.in :
79:    SEQ
80:      --          input string characters
81:      ch := 0
82:      open.file ("text","r",str.in)
83:      SEQ i = [ 0 FOR patlen -1]
84:        str.in ? ch
85:      WHILE ch <> (-1)
86:        SEQ
87:          --          I/O operations.
88:          str.in ? ch
89:          --          calculations.
90:          PAR
91:            s.out ! ch
92:            PAR i = [0 for 2*patlen ]
93:              ctl.out[i] ! ch
94:          close.file (str.in):
95:
96:  --
97:  ---***** FAN-IN cell definition *****
98:  --
99:  --Gets as input two single character comparison results and then
100:  --summed them up using the AND operation before outputting the
101:  --corresponding result.
102:  --
103:  PROC fan.in (CHAN r.in1, r.in2, r.out, ctl.in ) =
104:    VAR ctl, r, r1, r2 :
105:    SEQ
106:      ctl := 0
107:      r := FALSE
108:      WHILE ctl <> (-1)
109:        SEQ
110:          --          I/O operations
111:          PAR
112:            ctl.in ? ctl
113:            r.in1 ? r1
114:            r.in2 ? r2
115:            r.out ! r
116:          --          calculation.
117:          r := r1 AND r2 :
118:
119:  --
120:  ---***** SINK definition *****
121:  --
122:  --
123:  -- Gets as input a signal from the bottom FAN-IN cell in the
124:  -- the tree structure. If the signal is true ( a successful
125:  -- search ) then the exact position of the current pattern
126:  -- occurrence in the input string is ouput.
127:  --
128:  PROC sink ( CHAN r.in, s.in, ctl.in) =
129:    VAR ch, ctl, r, chpos:
130:    SEQ

```

```

131:      chpos:= -(lopat+1)
132:      ctl := 0
133:      WHILE ctl <> (-1)
134:          SEQ
135:              --                      I/O operations.
136:              PAR
137:                  chpos:=chpos+1
138:                  ctl.in ? ctl
139:                  r.in? r
140:                  s.in ? ch
141:              --                      Calculations.
142:              IF
143:                  r
144:                  put (chpos," Pattern found at " ) :
145:      --
146:      -----***** SYSTEM configuration -----*****
147:      --
148:      -- The system is specified by indicating the corresponding
149:      -- channels that link source, ips's ,fan-in and sink to form
150:      -- the solution network .
151:      --
152:      PROC system =
153:          PAR
154:              source.string ( s.c[0], ctl.c )
155:              PAR i = [0 FOR patlen ]
156:                  ips ( i, s.c[i], s.c[i+1], r.c[i], ctl.c[i] )
157:              PAR j = [0 FOR patlen-1 ]
158:                  fan.in (r.c[2*j],r.c[(2*j)+1],r.c[patlen+j],
159:                          ctl.c[patlen+j] )
160:                  sink ( r.c[2*(patlen-1)],s.c[patlen],ctl.c[(2*patlen)-1] ):
161:      --
162:      -----***** Pattern input procedure -----*****
163:      --
164:      PROC inp.pattern =
165:          VAR ch :
166:          SEQ
167:              --                      input pattern characters
168:              str.to.screen ( " Input pattern " )
169:              patlen:= 0
170:              keyboard ? ch
171:              WHILE ch <> '*n'
172:                  SEQ
173:                      patlen:= patlen+1
174:                      pat[BYTE patlen] := ch
175:                      screen ! pat[BYTE patlen]
176:                      keyboard ? ch
177:              put (patlen," Please enter Log2 of this value : ")
178:              get (lopat, " Thank you ")
179:              pat[BYTE 0]:= patlen :
180:      --
181:      --
182:      -----***** string input procedure -----*****
183:      --
184:      --This procedure open the string file and read the first patlen
185:      --characters in str which is used at the initialisation phase
186:      --of the ips's.
187:      PROC inp.string =
188:          CHAN str.in :
189:          SEQ
190:              open.file ("text","r",str.in)
191:              SEQ i = [0 for patlen]
192:                  str.in ? str [ BYTE i+1]
193:              str[ BYTE 0 ] := patlen.
194:              close.file (str.in) :
195:

```

```
196:  --
197:  --***** MAIN program *****
198:  --
199:  SEQ
200:      inp.pattern
201:      inp.string
202:      system
```

